7 KNOWLEDGE REPRESENTATION

This grand book, the universe, ... is written in the language of mathematics, and its characters are triangles, circles, and other geometric figures without which it is humanly impossible to understand a word of it; without these one wanders about in a dark labyrinth. . .

—GALILEO GALILEI, *Discorsi E Dimonstrazioni Matematiche Introno a Due Nuove Scienze (1638)*

Since no organism can cope with infinite diversity, one of the basic functions of all organisms is the cutting up of the environment into classifications by which non-identical stimuli can be treated as equivalent. . .

—ELEANOR ROSCH, *Principles of Categorization (1978)*

We have always two universes of discourse—call them "physical" and "phenomenal", or what you will—one dealing with questions of quantitative and formal structure, the other with those qualities that constitute a "world". All of us have our own distinctive mental worlds, our own inner journeyings and landscapes, and these, for most of us, require no clear neurological "correlate".

—OLIVER SACKS, *The Man Who Mistook His Wife for a Hat (1987)*

7.0 Issues in Knowledge Representation

The representation of information for use in intelligent problem solving offers important and difficult challenges that lie at the core of AI. In Section 7.1, we present a brief historical retrospective of early research in representation; topics include *semantic networks*, *conceptual dependencies*, *scripts*, and *frames*. Section 7.2 offers a more modern representation used in natural language programs, John Sowa's *conceptual graphs*. In Section 7.3, we critique the requirement of creating centralized and explicit representational schemes. Brooks' alternative is the *subsumption architecture* for robots. Section 7.4 presents *agents*,

another alternative to centralized control. In later chapters we extend our discussion of representations to include the stochastic (Section 9.3 and Chapter 13), the connectionist (Chapter 11), and the genetic/emergent (Chapter 12).

We begin our discussion of representation from an historical perspective, where a knowledge base is described as a mapping between the objects and relations in a problem domain and the computational objects and relations of a program (Bobrow 1975). The results of inferences in the knowledge base are assumed to correspond to the results of actions or observations in the world. The computational objects, relations, and inferences available to programmers are mediated by the knowledge representation language.

There are general principles of knowledge organization that apply across a variety of domains and can be directly supported by a representation language. For example, class hierarchies are found in both scientific and commonsense classification systems. How may we provide a general mechanism for representing them? How may we represent definitions? Exceptions? When should an intelligent system make default assumptions about missing information and how can it adjust its reasoning should these assumptions prove wrong? How may we best represent time? Causality? Uncertainty? Progress in building intelligent systems depends on discovering the principles of knowledge organization and supporting them through higher-level representational tools.

It is useful to distinguish between a representational *scheme* and the *medium* of its implementation. This is similar to the distinction between data structures and programming languages. Programming languages are the *medium* of implementation; the data structure is the *scheme*. Generally, knowledge representation languages are more constrained than the predicate calculus or programming languages. These constraints take the form of explicit structures for representing categories of knowledge. Their medium of implementation might be Prolog, Lisp, or more common languages such as C++ or Java.

Our discussion to this point illustrates the traditional view of AI representational schemes, a view that often includes a global knowledge base of language structures reflecting a static and "preinterpreted bias" of a "real world". More recent research in robotics (Brooks 1991a, Lewis and Luger 2000), situated cognition (Agre and Chapman 1987, Lakoff and Johnson 1999), agent-based problem solving (Jennings et al. 1998, Wooldridge 2000; Fatima et al. 2005, 2006; Vieira et al. 2007), and philosophy (Clark 1997) has challenged this traditional approach. These problem domains require distributed knowledge, a world that can itself be used as a partial knowledge structure, the ability to reason with partial information, and even representations that can themselves evolve as they come to experience the invariants of a problem domain. These approaches are introduced in Sections 7.3 and 7.4.

7.1 A Brief History of AI Representational Schemes

7.1.1 Associationist Theories of Meaning

Logical representations grew out of the efforts of philosophers and mathematicians to characterize the principles of correct reasoning. The major concern of logic is the development of formal representation languages with sound and complete inference rules. As a

result, the semantics of predicate calculus emphasizes *truth-preserving* operations on well-formed expressions. An alternative line of research has grown out of the efforts of psychologists and linguists to characterize the nature of human understanding. This work is less concerned with establishing a science of correct reasoning than with describing the way in which humans actually acquire, associate, and use knowledge of their world. This approach has proved particularly useful to the AI application areas of natural language understanding and commonsense reasoning.

There are many problems that arise in mapping commonsense reasoning into formal logic. For example, it is common to think of the operators $\vee$ and $\rightarrow$ as corresponding to the English "or" and "if ... then ...". However, these operators in logic are concerned solely with truth values and ignore the fact that the English "if ... then ..." suggests specific relationship (often more coorelational than causal) between its premises and its conclusion. For example, the sentence "If a bird is a cardinal then it is red" (associating the bird cardinal with the color red) can be written in predicate calculus:

$$\forall\ X\ (\text{cardinal}(X) \rightarrow \text{red}(X)).$$

This may be changed, through a series of truth-preserving operations, Chapter 2, into the logically equivalent expression

$$\forall\ X\ (\neg\ \text{red}(X) \rightarrow \neg\ \text{cardinal}(X)).$$

These two expressions have the same truth value; that is, the second is true if and only if the first is true. However, truth value equivalence is inappropriate in this situation. If we were to look for physical evidence of the truth of these statements, the fact that this sheet of paper is not red and also not a cardinal is evidence for the truth of the second expression. Because the two expressions are logically equivalent, it follows that it is also evidence for the truth of the first statement. This leads to the conclusion that the whiteness of the sheet of paper is evidence that cardinals are red.

This line of reasoning strikes us as meaningless and rather silly. The reason for this incongruity is that logical implication only expresses a relationship between the truth values of its operands, while the English sentence implied a positive coorelation between membership in a class and the possession of properties of that class. In fact, the genetic makeup of a bird causes it to have a certain color. This relationship is lost in the second version of the expression. Although the fact that the paper is not red is consistent with the truth of both sentences, it is irrelevant to the causal nature of the color of birds.

Associationist theories, following the empiricist tradition in philosophy, define the meaning of an object in terms of a network of associations with other objects. For the associationist, when humans perceive an object, that perception is first mapped into a concept. This concept is part of our entire knowledge of the world and is connected through appropriate relationships to other concepts. These relationships form an understanding of the properties and behavior of objects such as snow. For example, through experience, we associate the concept snow with other concepts such as cold, white, snowman, slippery, and ice. Our understanding of snow and the truth of statements such as "snow is white" and "the snowman is white" manifests itself out of this network of associations.

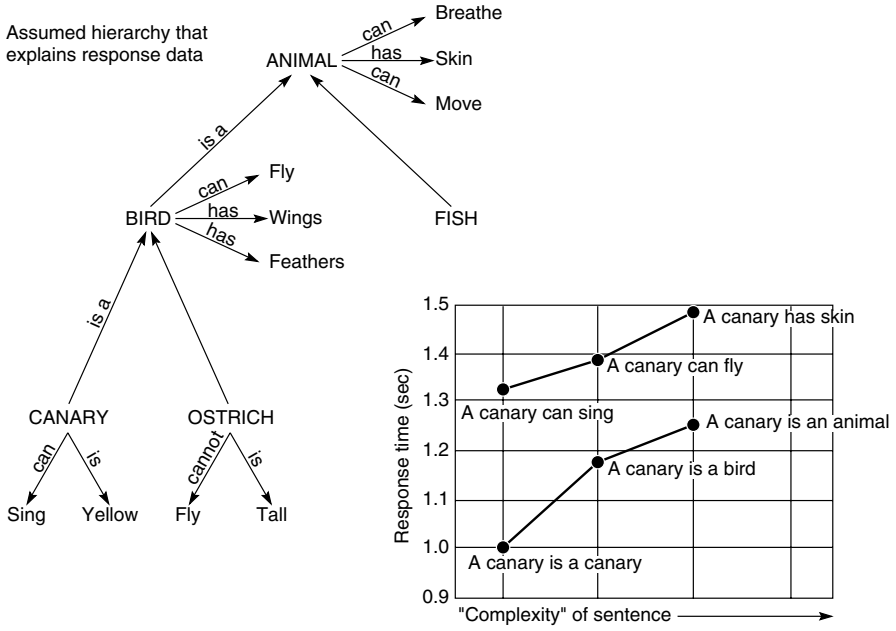


Figure 7.1 Semantic network developed by Collins and Quillian in their research on human information storage and response times (Harmon and King 1985).

There is psychological evidence that in addition to their ability to associate concepts, humans also organize their knowledge hierarchically, with information kept at the highest appropriate levels of the taxonomy. Collins and Quillian (1969) modeled human information storage and management using a semantic network (Figure 7.1). The structure of this hierarchy was derived from laboratory testing of human subjects. The subjects were asked questions about different properties of birds, such as, "Is a canary a bird?" or "Can a canary sing?" or "Can a canary fly?".

As obvious as the answers to these questions may seem, reaction-time studies indicated that it took longer for subjects to answer "Can a canary fly?" than it did to answer "Can a canary sing?" Collins and Quillian explain this difference in response time by arguing that people store information at its most abstract level. Instead of trying to recall that canaries fly, and robins fly, and swallows fly, all stored with the individual bird, humans remember that canaries are birds and that birds have (usually) the property of flying. Even more general properties such as eating, breathing, and moving are stored at the "animal" level, and so trying to recall whether a canary can breathe should take longer than recalling whether a canary can fly. This is, of course, because the human must travel further up the hierarchy of memory structures to get the answer.

The fastest recall was for the traits specific to the bird, say, that it can sing or is yellow. Exception handling also seemed to be done at the most specific level. When subjects were asked whether an ostrich could fly, the answer was produced faster than when they were asked whether an ostrich could breathe. Thus the hierarchy ostrich → bird → animal

seems not to be traversed to get the exception information: it is stored directly with ostrich. This knowledge organization has been formalized in inheritance systems.

Inheritance systems allow us to store information at the highest level of abstraction, which reduces the size of knowledge bases and helps prevent update inconsistencies. For example, if we are building a knowledge base about birds, we can define the traits common to all birds, such as flying or having feathers, for the general class bird and allow a particular species of bird to inherit these properties. This reduces the size of the knowledge base by requiring us to define these essential traits only once, rather than requiring their assertion for every individual. Inheritance also helps us to maintain the consistency of the knowledge base when adding new classes and individuals. Assume that we are adding the species robin to an existing knowledge base. When we assert that robin is a subclass of songbird; robin inherits all of the common properties of both songbirds and birds. It is not up to the programmer to remember (or forget!) to add this information.

Graphs, by providing a means of explicitly representing relations using arcs and nodes, have proved to be an ideal vehicle for formalizing associationist theories of knowledge. A *semantic network* represents knowledge as a graph, with the nodes corresponding to facts or concepts and the arcs to relations or associations between concepts. Both nodes and links are generally labeled. For example, a semantic network that defines the properties of snow and ice appears in Figure 7.2. This network could be used (with appropriate inference rules) to answer a range of questions about snow, ice, and snowmen. These inferences are made by following the links to related concepts. Semantic networks also implement inheritance; for example, frosty inherits all the properties of snowman.

The term "semantic network" encompasses a family of graph-based representations. These differ chiefly in the names that are allowed for nodes and links and the inferences that may be performed. However, a common set of assumptions and concerns is shared by all network representation languages; these are illustrated by a discussion of the history of network representations. In Section 7.2 we examine *conceptual graphs* (Sowa 1984), a more modern network representation language that integrates many of these ideas.

7.1.2 Early Work in Semantic Nets

Network representations have almost as long a history as logic. The Greek philosopher Porphyry created tree-based type hierarchies - with their roots at the top - to describe Aristotle's categories (Porphyry 1887). Frege developed a tree notation for logic expressions. Perhaps the earliest work to have a direct influence on contemporary semantic nets was Charles S. Peirce's system of existential graphs, developed in the nineteenth century (Roberts 1973). Peirce's theory had all the expressive power of first-order predicate calculus, with an axiomatic basis and formal rules of inference.

Graphs have long been used in psychology to represent structures of concepts and associations. Selz (1913, 1922) pioneered this work, using graphs to represent concept hierarchies and the inheritance of properties. He also developed a theory of schematic anticipation that influenced AI work in frames and schemata. Anderson, Norman, Rumelhart, and others have used networks to model human memory and intellectual performance (Anderson and Bower 1973, Norman et al. 1975).

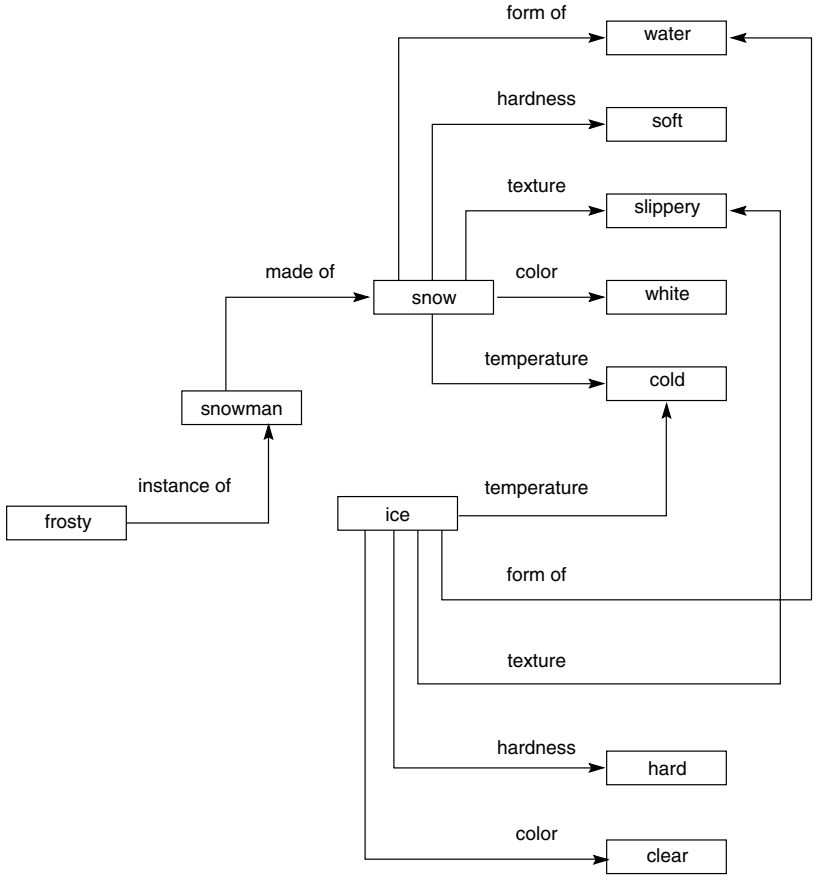


Figure 7.2 Network representation of properties of snow and ice.

Much of the research in network representations has been done in the arena of natural language understanding. In the general case, language understanding requires an understanding of common sense, the ways in which physical objects behave, the interactions that occur between humans, and the ways in which human institutions are organized. A natural language program must understand intentions, beliefs, hypothetical reasoning, plans, and goals. Because of these requirements language understanding has always been a driving force for research in knowledge representation.

The first computer implementations of semantic networks were developed in the early 1960s for use in machine translation. Masterman (1961) defined a set of 100 primitive concept types and used them to define a dictionary of 15,000 concepts. Wilks (1972) continued to build on Masterman's work in semantic network-based natural language systems. Shapiro's (1971) MIND program was the first implementation of a propositional calculus based semantic network. Other early AI workers exploring network representations include Ceccato (1961), Raphael (1968), Reitman (1965), and Simmons (1966).

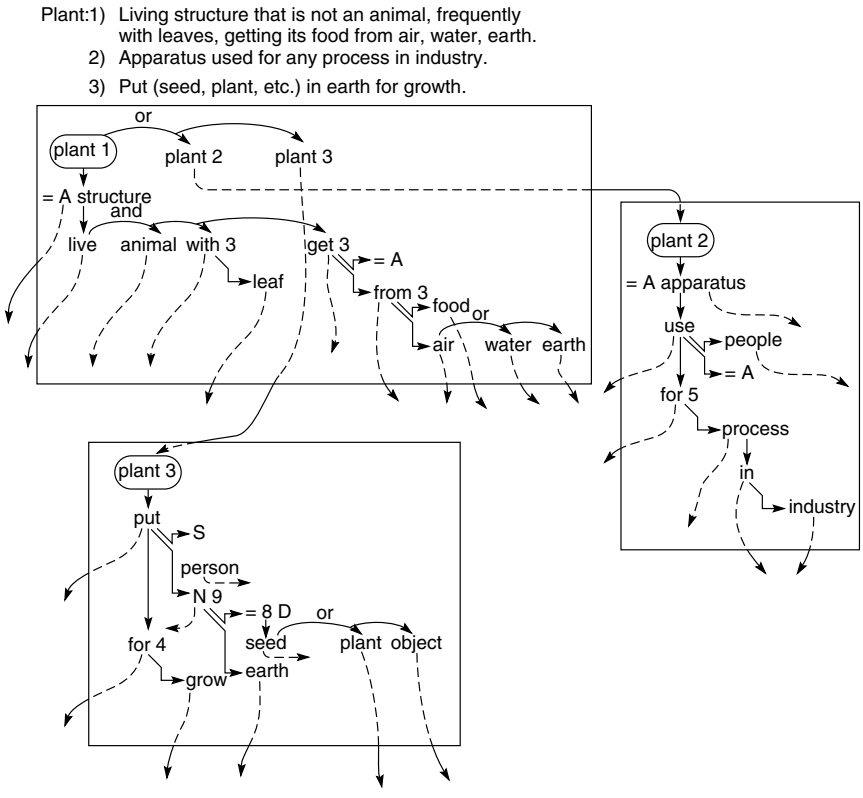


Figure 7.3 Three planes representing three definitions of the word plant (Quillian 1967).

An influential program that illustrates many of the features of early semantic networks was written by Quillian in the late 1960s (Quillian 1967). This program defined English words in much the same way that a dictionary does: a word is defined in terms of other words, and the components of the definition are defined in the same fashion. Rather than formally defining words in terms of basic axioms, each definition simply leads to other definitions in an unstructured and sometimes circular fashion. In looking up a word, we traverse this "network" until we are satisfied that we understand the original word.

Each node in Quillian's network corresponded to a *word concept*, with associative links to other word concepts that formed its definition. The knowledge base was organized into *planes*, where each plane was a graph that defined a single word. Figure 7.3, taken from a paper by Quillian (1967), illustrates three planes that capture three different definitions of the word "plant:" a living organism (plant 1), a place where people work (plant 2), and the act of putting a seed in the ground (plant 3).

The program used this knowledge base to find relationships between pairs of English words. Given two words, it would search the graphs outward from each word in a breadth-first fashion, searching for a common concept or *intersection node*. The paths to

this node represented a relationship between the word concepts. For example, Figure 7.4 from the same paper, shows the *intersection paths* between cry and comfort.

Using this intersection path, the program was able to conclude:

> cry 2 is among other things to make a sad sound. To comfort 3 can be to make 2 something less sad (Quillian 1967).

The numbers in the response indicate that the program has selected from among different meanings of the words.

Quillian (1967) suggested that this approach to semantics might provide a natural language understanding system with the ability to:

1. Determine the meaning of a body of English text by building up collections of these intersection nodes.
2. Choose between multiple meanings of words by finding the meanings with the shortest intersection path to other words in the sentence. For example, it could select a meaning for "plant" in "Tom went home to water his new plant" based on the intersection of the word concepts "water" and "plant."
3. Answer a flexible range of queries based on associations between word concepts in the queries and concepts in the system.

Although this and other early work established the power of graphs to model associative meaning, it was limited by the extreme generality of the formalism. Knowledge was generally structured in terms of specific relationships such as object/property, class/subclass, and agent/verb/object.

7.1.3 Standardization of Network Relationships

In itself, a graph notation of relationships has little computational advantage over logic; it is just another notation for relationships between objects. In fact SNePS, Semantic Net Processing System (Shapiro 1979, Shapiro et al. 2006), was a theorem prover in the first-order predicate calculus. The power of his network representations comes from the definition of links and associated inference rules such as inheritance.

Though Quillian's early work established most of the significant features of the semantic network formalism, such as labeled arcs and links, hierarchical inheritance, and inferences along associational links, it proved limited in its ability to deal with the complexities of many domains. One of the main reasons for this failure was the poverty of relationships (links) that captured the deeper semantic aspects of knowledge. Most of the links represented extremely general associations between nodes and provided no real basis for the structuring of semantic relationships. The same problem is encountered in efforts to use predicate calculus to capture semantic meaning. Although the formalism is highly expressive and can represent almost any kind of knowledge, it is too unconstrained and places the burden of constructing appropriate sets of facts and rules on the programmer.

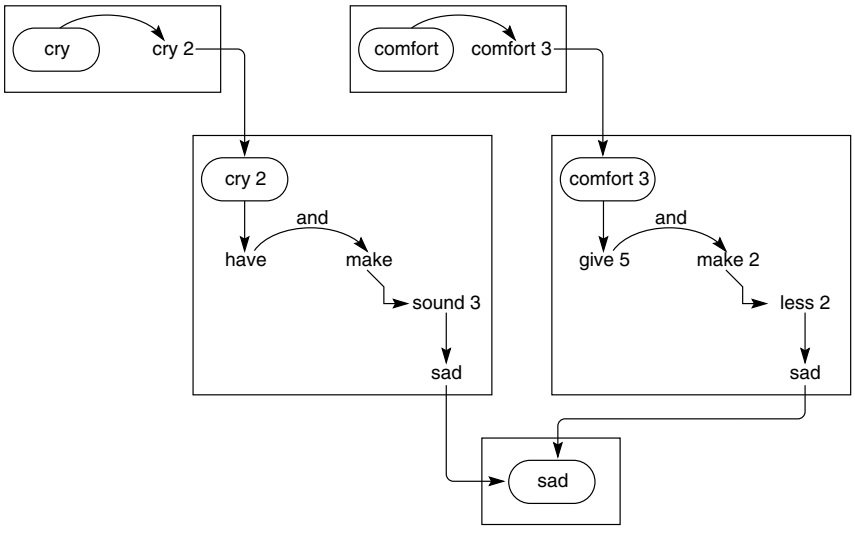


Figure 7.4 Intersection path between cry and comfort (Quillian 1967).

Much of the work in network representations that followed Quillian's focused on defining a richer set of link labels (relationships) that would more fully model the semantics of natural language. By implementing the fundamental semantic relationships of natural language as part of the *formalism*, rather than as part of the *domain knowledge* added by the system builder, knowledge bases require less handcrafting and achieve greater generality and consistency.

Brachman (1979) has stated:

> The key issue here is the isolation of the *primitives* for semantic network languages. The primitives of a network language are those things that the interpreter is programmed in advance to understand, and that are not usually represented in the network language itself.

Simmons (1973) addressed this need for standard relationships by focusing on the *case structure* of English verbs. In this verb-oriented approach, based on earlier work by Fillmore (1968), links define the roles played by nouns and noun phrases in the action of the sentence. Case relationships include *agent*, *object*, *instrument*, *location*, and *time*. A sentence is represented as a verb node, with various case links to nodes representing other participants in the action. This structure is called a *case frame*. In parsing a sentence, the program finds the verb and retrieves the case frame for that verb from its knowledge base. It then binds the values of the agent, object, etc., to the appropriate nodes in the case frame. Using this approach, the sentence "Sarah fixed the chair with glue" might be represented by the network in Figure 7.5.

Thus, the representation language itself captures much of the deep structure of natural language, such as the relationship between a verb and its subject (the agent relation) or

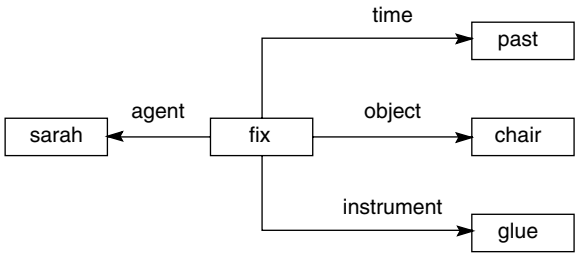


Figure 7.5 Case frame representation of the sentence Sarah fixed the chair with glue.

that between a verb and its object. Knowledge of the case structure of the English language is part of the network formalism itself. When the individual sentence is parsed, these built-in relationships indicate that Sarah is the person doing the fixing and that glue is used to put the chair together. Note that *these linguistic relationships are stored in a fashion that is independent of the actual sentence or even the language in which the sentence was expressed.* A similar approach was also taken in network languages proposed by Norman (1972) and Rumelhart et al. (1972, 1973).

A number of major research endeavors attempted to standardize link names even further (Masterman 1961, Wilks 1972, Schank and Colby 1973, Schank and Nash-Webber 1975). Each effort worked to establish a complete set of primitives that could be used to represent the semantic structure of natural language expressions in a uniform fashion. These were intended to assist in reasoning with language constructs and to be independent of the idiosyncrasies of individual languages or phrasing.

Perhaps the most ambitious attempt to model formally the deep semantic structure of natural language is Roger Schank's *conceptual dependency* theory (Schank and Rieger 1974). Conceptual dependency theory offers a set of four primitive conceptualizations from which the world of meaning is built. These are equal and independent. They are:

ACTs	actions
PPs	objects (picture producers)
AAs	modifiers of actions (action aiders)
PAs	modifiers of objects (picture aiders)

For example, all actions are assumed to reduce to one or more of the primitive ACTs. The primitives listed below are taken as the basic components of action, with more specific verbs being formed through their modification and combination.

ATRANS	transfer a relationship (give)
PTRANS	transfer physical location of an object (go)
PROPEL	apply physical force to an object (push)
MOVE	move body part by owner (kick)
GRASP	grab an object by an actor (grasp)

INGEST	ingest an object by an animal (eat)
EXPEL	expel from an animal s body (cry)
MTRANS	transfer mental information (tell)
MBUILD	mentally make new information (decide)
CONC	conceptualize or think about an idea (think)
SPEAK	produce sound (say)
ATTEND	focus sense organ (listen)

These primitives are used to define *conceptual dependency relationships* that describe meaning structures such as case relations or the association of objects and values. Conceptual dependency relationships are *conceptual syntax rules* and constitute a grammar of meaningful semantic relationships. These relationships can be used to construct an internal representation of an English sentence. A list of basic conceptual dependencies (Schank and Rieger 1974) appears in Figure 7.6. These capture, their creators feel, the fundamental semantic structures of natural language. For example, the first conceptual dependency in Figure 7.6 describes the relationship between a subject and its verb, and the third describes the verb–object relation. These can be combined to represent a simple transitive sentence such as "John throws the ball" (see Figure 7.7).

PP ⇔ ACT	indicates that an actor acts.
PP ⇔ PA	indicates that an object has a certain attribute.
ACT ←(O) PP	indicates the object of an action.
ACT ←(R) PP / PP	indicates the recipient and the donor of an object within an action.
ACT ←(D) PP / PP	indicates the direction of an object within an action.
ACT ←(1) ⇕	indicates the instrumental conceptualization for an action.
X ⇑ Y	indicates that conceptualization X caused conceptualization Y. When written with a C this form denotes that X COULD cause Y.
PP ⇐ PA2 / PA1	indicates a state change of an object.
PP1 ← PP2	indicates that PP2 is either PART OF or the POSSESSOR OF PP1.

Figure 7.6 Conceptual dependencies (Schank and Rieger 1974).

Figure 7.7 Conceptual dependency representation of the sentence John throws the ball.

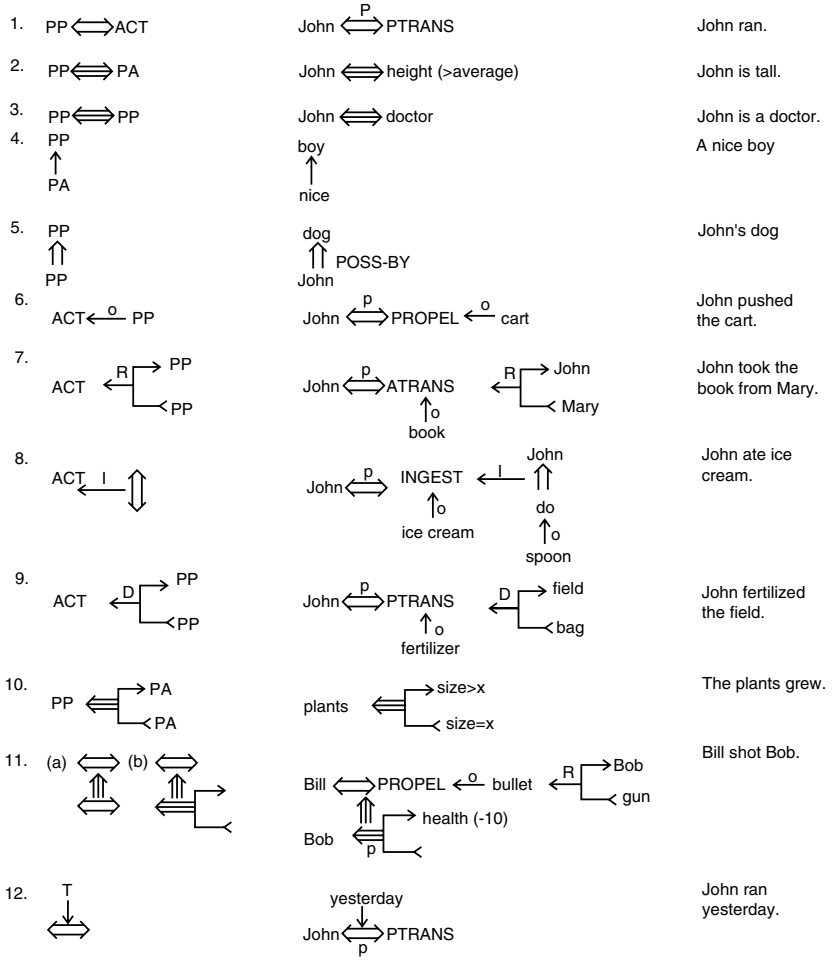


Figure 7.8 Some basic conceptual dependencies and their use in representing more complex English sentences, adapted from Schank and Colby (1973).

Finally, tense and mode information may be added to the set of conceptualizations. Schank supplies a list of attachments or modifiers to the relationships. A partial list of these includes:

p	past
f	future
t	transition
k	continuing
t_s	start transition
?	interrogative
t_f	finish transition
c	conditional
/	negative
nil	present
delta?	timeless

These relations are the first-level constructs of the theory, the simplest semantic relationships out of which more complex structures can be built. Further examples of how these basic conceptual dependencies can be composed to represent the meaning of simple English sentences appear in Figure 7.8.

Based on these primitives, the English sentence "John ate the egg" is represented as shown in Figure 7.9, where the symbols have the following meanings:

←	indicates the direction of dependency
⇔	indicates the agent—verb relationship
p	indicates past tense
INGEST	is a primitive act of the theory
O	object relation
D	indicates the direction of the object in the action

Another example of the structures that can be built using conceptual dependencies is the representation graph for "John prevented Mary from giving a book to Bill" (Figure 7.10). This particular example is interesting because it demonstrates how causality can be represented.

Conceptual dependency theory offers a number of important benefits. By providing a formal theory of natural language semantics, it reduces problems of ambiguity. Second, the representation itself directly captures much of natural language semantics, by attempting to provide a *canonical form* for the meaning of sentences. That is, all sentences that have the same meaning will be represented internally by *syntactically identical*, not just semantically equivalent, graphs. This canonical representation is an effort to simplify the inferences required for understanding. For example, we can demonstrate that two sentences mean the same thing with a simple match of conceptual dependency graphs; a representation that did not provide a canonical form might require extensive operations on differently structured graphs.

Unfortunately, it is questionable whether a program may be written to reliably reduce sentences to canonical form. As Woods (1985) and others have pointed out, reduction to a canonical form is provably uncomputable for monoids, a type of algebraic group that is far simpler than natural language. There is also no evidence that humans store their knowledge in any such sort of canonical form.

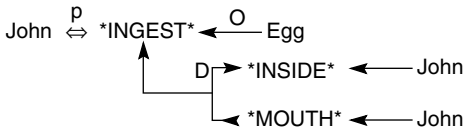


Figure 7.9 Conceptual dependency representing John ate the egg (Schank and Rieger 1974).

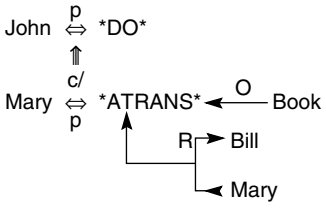


Figure 7.10 Conceptual dependency representation of the sentence John prevented Mary from giving a book to Bill (Schank and Rieger 1974).

Other criticisms of this point of view, besides objecting to the computational price paid in reducing everything to such low-level primitives, suggest that the primitives themselves are not adequate to capture many of the more subtle concepts that are important in natural language use. For example, the representation of "tall" in the second sentence of Figure 7.8 does not address the ambiguity of this term as fully as is done in systems such as fuzzy logic (Zadeh 1983 and Section 9.2.2).

However, no one can say that the conceptual dependency model has not been extensively studied and well understood. More than a decade of research guided by Schank has focused on refining and extending the model. Important extensions of conceptual dependencies include research in *scripts* and *memory organization packets*, or MOPs. The research in scripts examines the organization of knowledge in memory and the role this organization plays in reasoning, see Section 7.1.4. MOPs provided one of the supporting research areas for the design of case-based reasoners, Section 8.3. Conceptual dependency theory is a fully developed model of natural language semantics with consistency of purpose and wide applicability.

7.1.4 Scripts

A natural language understanding program must use a large amount of supporting knowledge to understand even the simplest conversation (Section 15.0). There is evidence that humans organize this knowledge into structures corresponding to typical situations (Bartlett 1932). If we are reading a story about restaurants, baseball, or politics, we resolve any ambiguities in the text in a way consistent with restaurants, baseball, or politics. If the subject of a story changes abruptly, there is evidence that people pause briefly in their

reading, presumably to change knowledge structures. It is hard to understand a poorly organized or structured story, possibly because we cannot easily fit it into any of our existing knowledge structures. There can also be errors in understanding when the subject of a conversation changes abruptly, presumably because we are confused over which context to use in resolving pronoun references and other ambiguities in the conversation.

A *script* is a structured representation describing a stereotyped sequence of events in a particular context. The script was originally designed by Schank and his research group (Schank and Abelson 1977) as a means of organizing *conceptual dependency* structures into descriptions of typical situations. Scripts are used in natural language understanding systems to organize a knowledge base in terms of the situations that the system is to understand.

Most adults are quite comfortable (i.e., they, as customers, know what to expect and how to act) in a restaurant. They are either met at the restaurant entrance or see some sign indicating that they should continue in and find a table. If a menu is not available at the table or if it is not presented by the waiter, then the customer will ask for one. Similarly, customers understand the routines for ordering food, eating, paying, and leaving.

In fact, the restaurant script is quite different from other eating scripts such as the "fast-food" model or the "formal family meal". In the fast-food model the customer enters, gets in line to order, pays for the meal (before eating), waits for a tray with the food, takes the tray, tries to find a clean table, and so on. These are two different stereotyped sequences of events, and each has a potential script.

The components of a script are:

Entry conditions or descriptors of the world that must be true for the script to be called. In our example script, these include an open restaurant and a hungry customer that has some money.

Results or facts that are true once the script has terminated; for example, the customer is full and poorer, the restaurant owner has more money.

Props or the "things" that support the content of the script. These will include tables, waiters, and menus. The set of props supports reasonable default assumptions about the situation: a restaurant is assumed to have tables and chairs unless stated otherwise.

Roles are the actions that the individual participants perform. The waiter takes orders, delivers food, and presents the bill. The customer orders, eats, and pays.

Scenes. Schank breaks the script into a sequence of scenes each of which presents a temporal aspect of the script. In the restaurant there is entering, ordering, eating, etc.

The elements of the script, the basic "pieces" of semantic meaning, are represented using conceptual dependency relationships. Placed together in a framelike structure, they represent a sequence of meanings, or an event sequence. The restaurant script taken from this research is presented in Figure 7.11.

The program reads a small story about restaurants and parses it into an internal conceptual dependency representation. Because the key concepts in this internal description match with the entry conditions of the script, the program binds the people and things

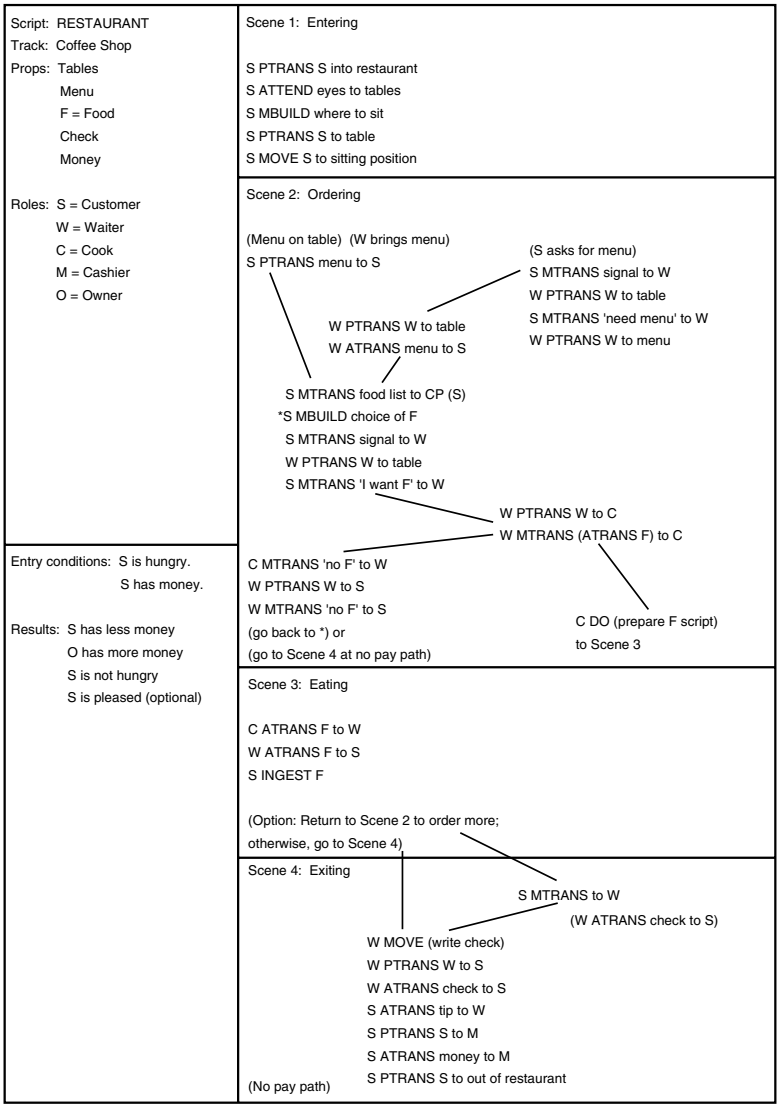


Figure 7.11 A restaurant script (Schank and Abelson 1977).

mentioned in the story to the roles and props mentioned in the script. The result is an expanded representation of the story contents, using the script to fill in any missing information and default assumptions. The program then answers questions about the story by referring to the script. The script allows the reasonable default assumptions that are essential to natural language understanding. For example:

EXAMPLE 7.1.1

John went to a restaurant last night. He ordered steak. When he paid he noticed he was running out of money. He hurried home since it had started to rain.

Using the script, the system can correctly answer questions such as: Did John eat dinner last night (the story only implied this)? Did John use cash or a credit card? How could John get a menu? What did John buy?

EXAMPLE 7.1.2

Amy Sue went out to lunch. She sat at a table and called a waitress, who brought her a menu. She ordered a sandwich.

Questions that might reasonably be asked of this story include: Why did the waitress bring Amy Sue a menu? Was Amy Sue in a restaurant (the example doesn't say so)? Who paid? Who was the "she" who ordered the sandwich? This last question is difficult. The most recently named female is the waitress, an incorrect assumption for pronoun reference. Script *roles* help to resolve such references and other ambiguities.

Scripts can also be used to interpret unexpected results or breaks in the scripted activity. Thus, in scene 2 of Figure 7.11 there is the choice point of "food" or "no food" delivered to the customer. This allows the following example to be understood.

EXAMPLE 7.1.3

Kate went to a restaurant. She was shown to a table and ordered sushi from the waitress. She sat there and waited for a long time. Finally, she got mad and left.

Questions that can be answered from this story using the restaurant script include: Who is the "she" who sat and waited? Why did she wait? Who was the "she" who got mad and left? Why did she get mad? Note that there are other questions that the script cannot answer, such as why people get upset/mad when the waiter does not come promptly? Like any knowledge-based system, scripts require the knowledge engineer to correctly anticipate the knowledge required.

Scripts, like frames and other structured representations, are subject to certain problems, including the script *match* problem and the *between-the-lines* problem. Consider Example 7.1.4, which could call either the *restaurant* or *concert* scripts. The choice is critical because "bill" can refer to either the restaurant check or the playbill of the concert.

EXAMPLE 7.1.4

John visited his favorite restaurant on the way to the concert. He was pleased by the bill because he liked Mozart.

Since script selection is usually based on matching "key" words, it is often difficult to determine which of two or more potential scripts should be used. The script match problem is "deep" in the sense that no algorithm exists for guaranteeing correct choices. It requires heuristic knowledge about the organization of the world, and perhaps even some backtracking; scripts assist only in the organization of that knowledge.

The between-the-lines problem is equally difficult: it is not possible to know ahead of time the possible occurrences that can break a script. For instance:

EXAMPLE 7.1.5

Melissa was eating dinner at her favorite restaurant when a large piece of plaster fell from the ceiling and landed on her date.

Questions: Was Melissa eating a date salad? Was Melissa's date plastered? What did she do next? As this example illustrates, structured representations can be inflexible. Reasoning can be locked into a single script, even though this may not be appropriate.

Memory organization packets (MOPs) address the problem of script inflexibility by representing knowledge as smaller components, MOPs, along with rules for dynamically combining them to form a schema that is appropriate to the current situation (Schank 1982). The organization of knowledge in memory is particularly important to implementations of case-based reasoning, in which the problem solver must efficiently retrieve a relevant prior problem solution from memory (Kolodner 1988a, Section 8.3).

The problems of organizing and retrieving knowledge are difficult and inherent to the modeling of semantic meaning. Eugene Charniak (1972) illustrated the amount of knowledge required to understand even simple children's stories. Consider a statement about a birthday party: Mary was given two kites for her birthday so she took one back to the store. We must know about the tradition of giving gifts at a party; we must know what a kite is and why Mary might not want two of them; we must know about stores and their exchange policies. In spite of these problems, programs using scripts and other semantic representations can understand natural language in limited domains. An example of this work is a program that interprets messages coming over the news wire services. Using scripts for natural disasters, coups, or other stereotypic stories, programs have shown remarkable success in this limited but realistic domain (Schank and Riesbeck 1981).

7.1.5 Frames

Another representational scheme, in many ways similar to scripts, that was intended to capture in explicitly organized data structures the implicit connections of information in a problem domain, was called *frames*. This representation supports the organization of knowledge into more complex units that reflect the organization of objects in the domain.

In a 1975 paper, Minsky describes a frame:

> Here is the essence of the frame theory: When one encounters a new situation (or makes a substantial change in one's view of a problem) one selects from memory a structure called a "frame". This is a remembered framework to be adapted to fit reality by changing details as necessary (Minsky 1975).

According to Minsky, a frame may be viewed as a static data structure used to represent well-understood stereotyped situations. Framelike structures seem to organize our

own knowledge of the world. We adjust to every new situation by calling up information structured by past experiences. We then specially fashion or revise the details of these past experiences to represent the individual differences for the new situation.

Anyone who has stayed in one or two hotel rooms has no trouble with entirely new hotels and their rooms. One expects to see a bed, a bathroom, a place to open a suitcase, a telephone, price and emergency evacuation information on the back of the entry door, and so on. The details of each room can be supplied when needed: color of the curtains, location and use of light switches, etc. There is also default information supplied with the hotel room frame: no sheets; call housekeeping; need ice: look down the hall; and so on. We do not need to build up our understanding for each new hotel room we enter. All of the pieces of a generic hotel room are organized into a conceptual structure that we access when checking into a hotel; the particulars of an individual room are supplied as needed.

We could represent these higher-level structures directly in a semantic network by organizing it as a collection of separate networks, each of which represents some stereotypic situation. Frames, as well as *object-oriented systems*, provide us with a vehicle for this organization, representing entities as structured objects with named slots and attached values. Thus a frame or schema is seen as a single complex entity.

For example, the hotel room and its components can be described by a number of individual frames. In addition to the bed, a frame could represent a chair: expected height is 20 to 40 cm, number of legs is 4, a default value, is designed for sitting. A further frame represents the hotel telephone: this is a specialization of a regular phone except that billing is through the room, there is a special hotel operator (default), and a person is able to use the hotel phone to get meals served in the room, make outside calls, and to receive other services. Figure 7.12 gives a frame representing the hotel room.

Each individual frame may be seen as a data structure, similar in many respects to the traditional "record", that contains information relevant to stereotyped entities. The slots in the frame contain information such as:

1. *Frame identification information*.
2. *Relationship of this frame to other frames*. The "hotel phone" might be a special instance of "phone," which might be an instance of a "communication device."
3. *Descriptors of requirements for a frame.* A chair, for instance, has its seat between 20 and 40 cm from the floor, its back higher than 60 cm, etc. These requirements may be used to determine when new objects fit the stereotype defined by the frame.
4. *Procedural information on use of the structure described*. An important feature of frames is the ability to attach procedural instructions to a slot.
5. *Frame default information*. These are slot values that are taken to be true when no evidence to the contrary has been found. For instance, chairs have four legs, telephones are pushbutton, or hotel beds are made by the staff.
6. *New instance information*. Many frame slots may be left unspecified until given a value for a particular instance or when they are needed for some aspect of problem solving. For example, the color of the bedspread may be left unspecified.

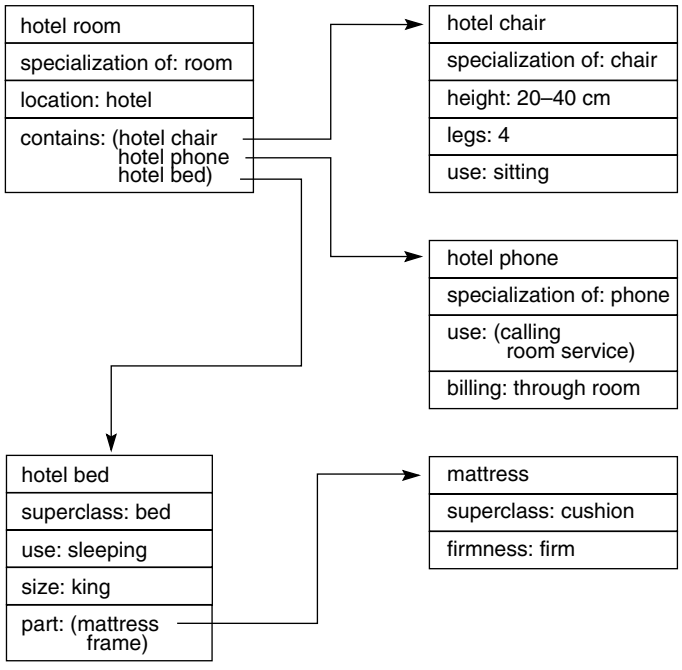


Figure 7.12 Part of a frame description of a hotel room. Specialization indicates a pointer to a superclass.

Frames extend semantic networks in a number of important ways. Although the frame description of hotel beds, Figure 7.12, might be equivalent to a network description, the frame version makes it much clearer that we are describing a bed with its various attributes. In the network version, there is simply a collection of nodes and we depend more on our interpretation of the structure to see the hotel bed as the primary object being described. This ability to organize our knowledge into such structures is an important attribute of a knowledge base.

Frames make it easier to organize our knowledge hierarchically. In a network, every concept is represented by nodes and links at the same level of specification. Very often, however, we may like to think of an object as a single entity for some purposes and only consider details of its internal structure for other purposes. For example, we usually are not aware of the mechanical organization of a car until something breaks down; only then do we pull up our "car engine schema" and try to find the problem.

Procedural attachment is an important feature of frames because it supports the linking of specific pieces of code to appropriate entities in the frame representation. For example, we might want to include the ability to generate graphic images in a knowledge base. A graphics language is more appropriate for this than a network language. We use procedural attachment to create *demons*. A demon is a procedure that is invoked as a side effect of some other action in the knowledge base. For example, we may wish the system to perform type checks or to run consistency tests whenever a certain slot value is changed.

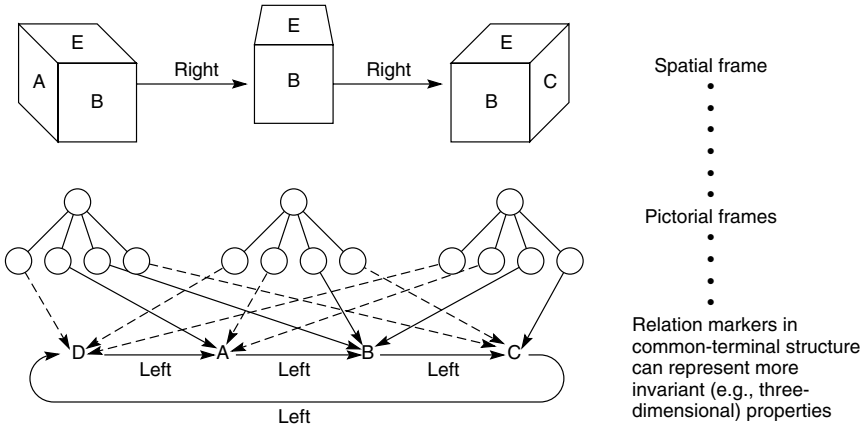


Figure 7.13 Spatial frame for viewing a cube (Minsky 1975).

Frame systems support class inheritance. The slots and default values of a class frame are inherited across the class/subclass and class/member hierarchy. For instance, a hotel phone could be a subclass of a regular phone except that (1) all out-of-building dialing goes through the hotel switchboard (for billing) and (2) hotel services may be dialed directly. Default values are assigned to selected slots to be used only if other information is not available: assume that hotel rooms have beds and are, therefore, appropriate places to go if you want to sleep; if you don't know how to dial the hotel front desk try "zero"; the phone may be assumed (no evidence to the contrary) to be pushbutton.

When an instance of the class frame is created, the system will attempt to fill its slots, either by querying the user, accepting the default value from the class frame, or executing some procedure or demon to obtain the instance value. As with semantic nets, slots and default values are inherited across a class/subclass hierarchy. Of course, default information can cause the data description of the problem to be nonmonotonic, letting us make assumptions about default values that may not always prove correct (see Section 9.1).

Minsky's own work on vision provides an example of frames and their use in default reasoning: the problem of recognizing that different views of an object actually represent the same object. For example, the three perspectives of the one cube of Figure 7.13 actually look quite different. Minsky (1975) proposed a frame system that recognizes these as views of a single object by inferring the hidden sides as default assumptions.

The frame system of Figure 7.13 represents four of the faces of a cube. The broken lines indicate that a particular face is out of view from that perspective. The links between the frames indicate the relations between the views represented by the frames. The nodes, of course, could be more complex if there were colors or patterns that the faces contained. Indeed, each slot in one frame could be a pointer to another entire frame. Also, because given information can fill a number of different slots (face E in Figure 7.13), there need be no redundancy in the information that is stored.

Frames add to the power of semantic nets by allowing complex objects to be represented as a single frame, rather than as a large network structure. This also provides a very

natural way to represent stereotypic entities, classes, inheritance, and default values. Although frames, like logic and network representations, are a powerful tool, many of the problems of acquiring and organizing a complicated knowledge base must still be solved by the programmer's skill and intuition. Finally, this MIT research of the 1970s, as well as similar work at Xerox Palo Alto Research Center, led to the "object-oriented" programming design philosophy as well as building important implementation languages, including Smalltalk, C++, and Java.

7.2 Conceptual Graphs: a Network Language

Following on the early research work in AI that developed representational schemes (Section 7.1) a number of network languages were created to model the semantics of natural language and other domains. In this section, we examine a particular formalism in detail, to show how, in this situation, the problems of representing meaning were addressed. John Sowa's *conceptual graphs* (Sowa 1984) is an example of a network representation language. We define the rules for forming and manipulating conceptual graphs and the conventions for representing classes, individuals, and relationships. In Section 15.3.2 we show how this formalism may be used to represent meaning in natural language understanding.

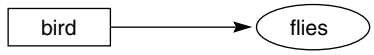
7.2.1 Introduction to Conceptual Graphs

A *conceptual graph* is a finite, connected, bipartite graph. The nodes of the graph are either *concepts* or *conceptual relations*. Conceptual graphs do not use labeled arcs; instead the conceptual relation nodes represent relations between concepts. Because conceptual graphs are bipartite, concepts only have arcs to relations, and vice versa. In Figure 7.14 dog and brown are concept nodes and color a conceptual relation. To distinguish these types of nodes, we represent concepts as boxes and conceptual relations as ellipses.

In conceptual graphs, concept nodes represent either concrete or abstract objects in the world of discourse. Concrete concepts, such as a cat, telephone, or restaurant, are characterized by our ability to form an image of them in our minds. Note that concrete concepts include generic concepts such as cat or restaurant along with concepts of specific cats and restaurants. We can still form an image of a generic cat. Abstract concepts include things such as love, beauty, and loyalty that do not correspond to images in our minds.

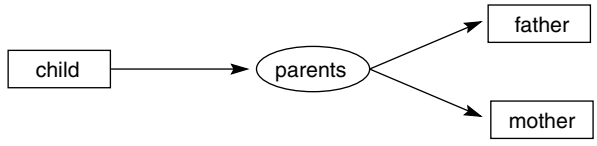
Conceptual relation nodes indicate a relation involving one or more concepts. One advantage of formulating conceptual graphs as bipartite graphs rather than using labeled arcs is that it simplifies the representation of relations of any arity. A relation of arity n is represented by a conceptual relation node having n arcs, as shown in Figure 7.14.

Each conceptual graph represents a single proposition. A typical knowledge base will contain a number of such graphs. Graphs may be arbitrarily complex but must be finite. For example, one graph in Figure 7.14 represents the proposition "A dog has a color of brown". Figure 7.15 is a graph of somewhat greater complexity that represents the sentence "Mary gave John the book". This graph uses conceptual relations to represent the



Flies is a 1-ary relation.

Color is a 2-ary relation.



Parents is a 3-ary relation.

Figure 7.14 Conceptual relations of different arities.

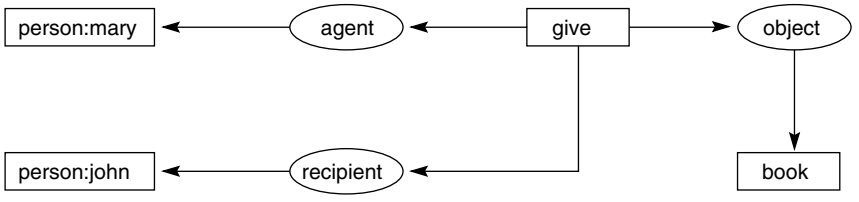


Figure 7.15 Graph of Mary gave John the book.

cases of the verb "to give" and indicates the way in which conceptual graphs are used to model the semantics of natural language.

7.2.2 Types, Individuals, and Names

Many early designers of semantic networks were careless in defining class/member and class/subclass relationships, with resulting semantic confusion. For example, the relation between an individual and its class is different from the relation between a class (such as dog) and its superclass (carnivore). Similarly, certain properties belong to individuals, and others belong to the class itself; the representation should provide a vehicle for making this distinction. The properties of having fur and liking bones belong to individual dogs; the class "dog" does not have fur or eat anything. Properties that are appropriate to the class include its name and membership in a zoological taxonomy.

Figure 7.16 Conceptual graph indicating that the dog named emma is brown.

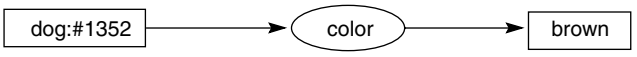


Figure 7.17 Conceptual graph indicating that a particular (but unnamed) dog is brown.

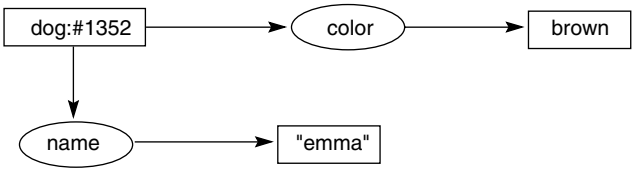


Figure 7.18 Conceptual graph indicating that a dog named emma is brown.

In conceptual graphs, every concept is a unique individual of a particular type. Each concept box is labeled with a *type* label, which indicates the class or type of individual represented by that node. Thus, a node labeled dog represents some individual of that type. Types are organized into a hierarchy. The type dog is a subtype of carnivore, which is a subtype of mammal, etc. Boxes with the same type label represent concepts of the same type; however, these boxes may or may not represent the same individual concept.

Each concept box is labeled with the names of the type and the individual. The type and individual labels are separated by a colon, ":". The graph of Figure 7.16 indicates that the dog "Emma" is brown. The graph of Figure 7.17 asserts that some unspecified entity of type dog has a color of brown. If the individual is not indicated, the concept represents an unspecified individual of that type.

Conceptual graphs also let us indicate specific but unnamed individuals. A unique token called a *marker* indicates each individual in the world of discourse. This marker is written as a number preceded by a #. Markers are different from names in that they are unique: individuals may have one name, many names, or no name at all, but they have exactly one marker. Similarly, different individuals may have the same name but may not have the same marker. This distinction gives us a basis for dealing with the semantic ambiguities that arise when we give objects names. The graph of Figure 7.17 asserts that a particular dog, #1352, is brown.

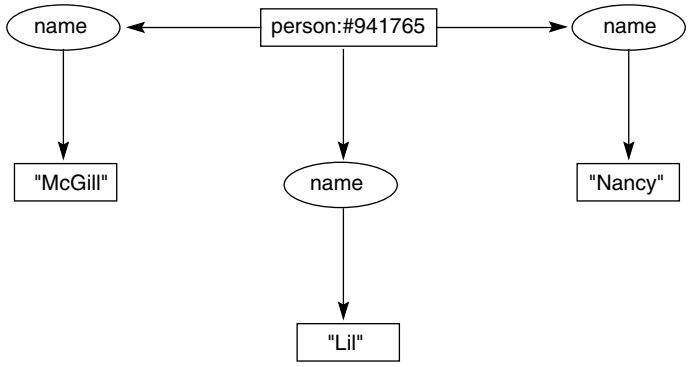


Figure 7.19 Conceptual graph of a person with three names.

Markers allow us to separate an individual from its name. If dog #1352 is named "Emma," we can use a conceptual relation called name to add this to the graph. The result is the graph of Figure 7.18. The name is enclosed in double quotes to indicate that it is a string. Where there is no danger of ambiguity, we may simplify the graph and refer to the individual directly by name. Under this convention, the graph of Figure 7.18 is equivalent to the graph of Figure 7.16.

Although we frequently ignore it both in casual conversation and in formal representations, this distinction between an individual and its name is an important one that should be supported by a representation language. For example, if we say that "John" is a common name among males, we are asserting a property of the name itself rather than of any individual named "John". This allows us to represent such English sentences as "'Chimpanzee' is the name of a species of primates". Similarly, we may want to represent the fact that an individual has several different names. The graph of Figure 7.19 represents the situation described in the song lyric: "Her name was McGill, and she called herself Lil, but everyone knew her as Nancy" (Lennon and McCartney 1968).

As an alternative to indicating an individual by its marker or name, we can also use the generic marker * to indicate an unspecified individual. By convention, this is often omitted from concept labels; a node given just a type label, dog, is equivalent to a node labeled dog:*. In addition to the generic marker, conceptual graphs allow the use of named variables. These are represented by an asterisk followed by the variable name (e.g., *X or *foo). This is useful if two separate nodes are to indicate the same, but unspecified, individual. The graph of Figure 7.20 represents the assertion "The dog scratches its ear with its paw". Although we do not know which dog is scratching its ear, the variable *X indicates that the paw and the ear belong to the same dog that is doing the scratching.

To summarize, each concept node can indicate an individual of a specified type. This individual is the *referent* of the concept. This reference is indicated either individually or generically. If the referent uses an individual marker, the concept is an *individual* concept; if the referent uses the generic marker, then the concept is *generic*. In Section 7.2.3 we present the conceptual graph type hierarchy.

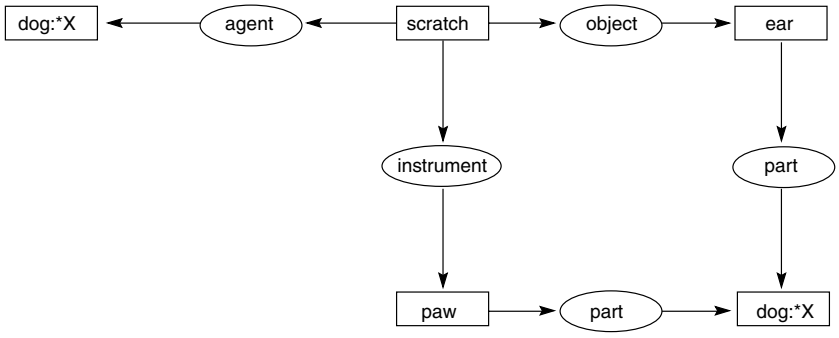


Figure 7.20 Conceptual graph of the sentence The dog scratches its ear with its paw.

7.2.3 The Type Hierarchy

The type hierarchy, as illustrated by Figure 7.21, is a partial ordering on the set of types, indicated by the symbol $\leq$. If s and t are types and $t \leq s$, then t is said to be a *subtype* of s and s is said to be a *supertype* of t. Because it is a partial ordering, a type may have one or more supertypes as well as one or more subtypes. If s, t, and u are types, with $t \leq s$ and $t \leq u$, then t is said to be a *common subtype* of s and u. Similarly, if $s \leq v$ and $u \leq v$ then v is a *common supertype* of s and u.

The type hierarchy of conceptual graphs forms a lattice, a common form of multiple inheritance system. In a lattice, types may have multiple parents and children. However, every pair of types must have a *minimal common supertype* and a *maximal common subtype*. For types s and u, v is a minimal common supertype if $s \leq v$, $u \leq v$, and for any w, a common supertype of s and u, $v \leq w$. Maximal common subtype has a corresponding definition. The minimal common supertype of a collection of types is the appropriate place to define properties common only to those types. Because many types, such as emotion and rock, have no obvious common supertypes or subtypes, it is necessary to add types that fill these roles. To make the type hierarchy a true lattice, conceptual graphs include two special types. The *universal type*, indicated by T, is a supertype of all types. The *absurd type*, indicated by $\perp$, is a subtype of all types.

7.2.4 Generalization and Specialization

The theory of conceptual graphs includes a number of operations that create new graphs from existing graphs. These allow for the generation of a new graph by either specializing or generalizing an existing graph, operations that are important for representing the semantics of natural language. The four operations are *copy, restrict, join*, and *simplify*, as seen in Figure 7.22. Assume that g_1 and g_2 are conceptual graphs. Then:

The *copy* rule allows us to form a new graph, g, that is the exact copy of g_1.

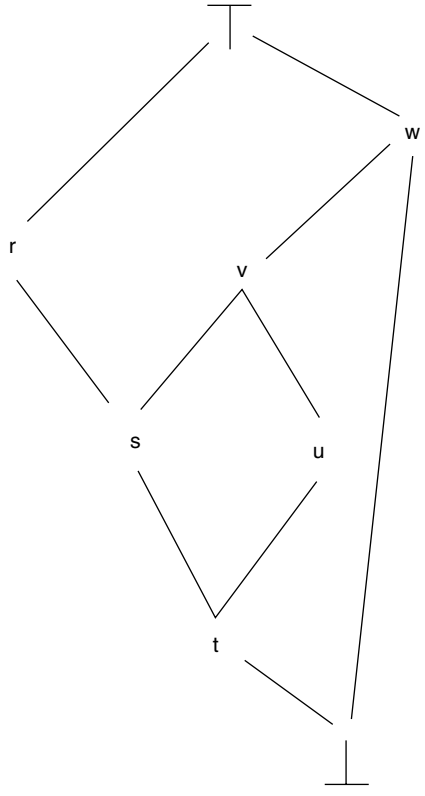


Figure 7.21 A type lattice illustrating subtypes, supertypes, the universal type, and the absurd type. Arcs represent the relationship.

Restrict allows concept nodes in a graph to be replaced by a node representing their specialization. There are two cases:

1. If a concept is labeled with a generic marker, the generic marker may be replaced by an individual marker.
2. A type label may be replaced by one of its subtypes, if this is consistent with the referent of the concept. In Figure 7.22 we can replace animal with dog.

The *join* rule lets us combine two graphs into a single graph. If there is a concept node c_1 in the graph s_1 that is identical to a concept node c_2 in s_2, then we can form a new graph by deleting c_2 and linking all of the relations incident on c_2 to c_1. *Join* is a specialization rule, because the resulting graph is less general than either of its components.

If a graph contains two duplicate relations, then one of them may be deleted, along with all its arcs. This is the *simplify* rule. Duplicate relations often occur as the result of a *join* operation, as in graph g_4 of Figure 7.22.

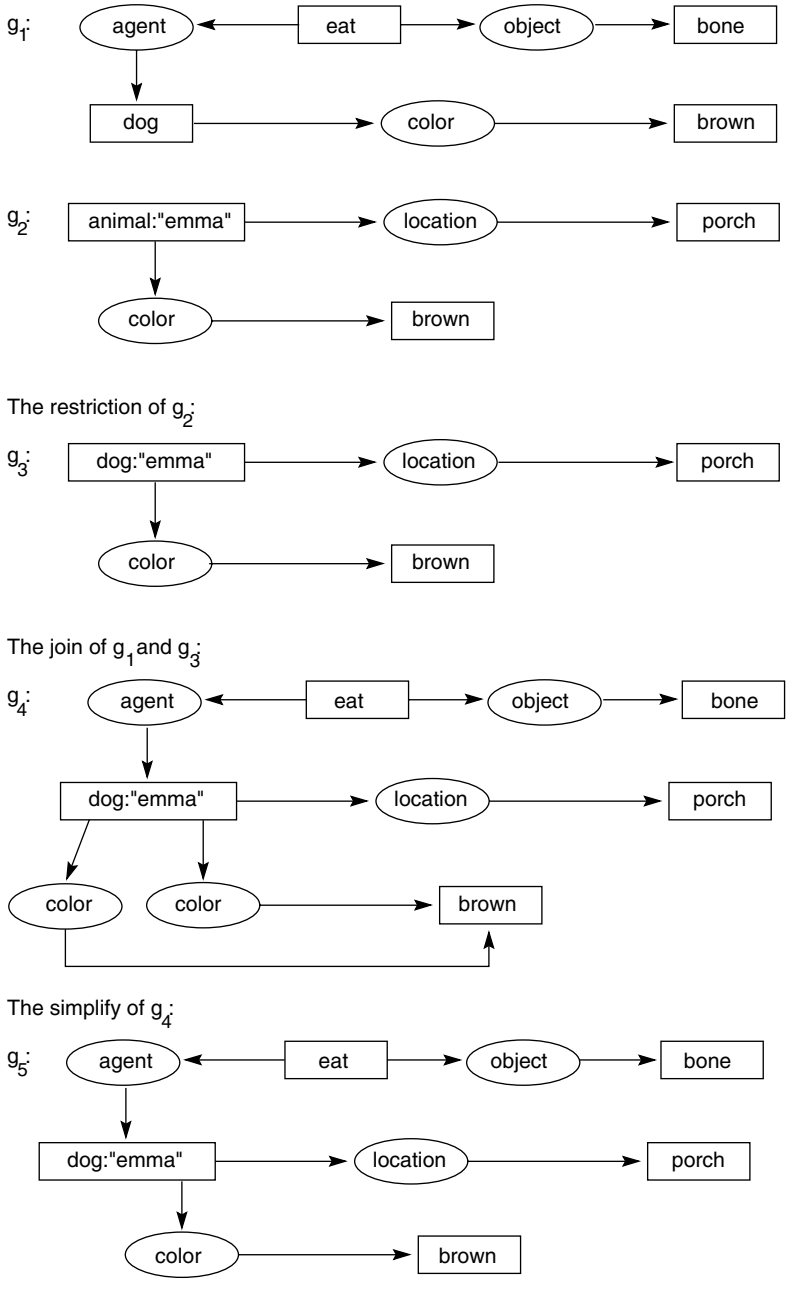


Figure 7.22 Examples of restrict, join, and simplify operations.

One use of the *restrict* rule is to make two concepts match so that a *join* can be performed. Together, *join* and *restrict* allow the implementation of inheritance. For example, the replacement of a generic marker by an individual implements the inheritance of the properties of a type by an individual. The replacement of a type label by a subtype label defines inheritance between a class and a superclass. By joining one graph to another and restricting certain concept nodes, we can implement inheritance of a variety of properties. Figure 7.23 shows how chimpanzees inherit the property of having a hand from the class primates by replacing a type label with its subtype. It also shows how the individual, Bonzo, inherits this property by instantiating a generic concept.

Similarly, we can use joins and restrictions to implement the plausible assumptions that play a role in common language understanding. If we are told that "Mary and Tom went out for pizza together," we automatically make a number of assumptions: they ate a round Italian bread covered with cheese and tomato sauce. They ate it in a restaurant and must have had some way of paying for it. This reasoning can be done using joins and restrictions. We form a conceptual graph of the sentence and then *join* it with the conceptual graphs (from our knowledge base) for pizzas and restaurants. The resulting graph lets us assume that they ate tomato sauce and paid their bill.

Join and *restrict* are specialization rules. They define a partial ordering on the set of derivable graphs. If a graph g_1 is a specialization of g_2, then we may say that g_2 is a generalization of g_1. Generalization hierarchies are important in knowledge representation. Besides providing the basis for inheritance and other commonsense reasoning schemes, generalization hierarchies are used in many learning methods, allowing us, for instance, to construct a generalized assertion from a particular training instance.

These rules are not rules of inference. They do not guarantee that true graphs will be derived from true graphs. For example, in the restriction of the graph of Figure 7.19, the result may not be true; Emma may be a cat. Similarly, the joining example of Figure 7.19 is not truth-preserving either: the dog on the porch and the dog that eats bones may be different animals. These operations are *canonical formation rules*, and although they do not preserve truth, they have the subtle but important property of preserving "meaningfulness". This is an important guarantee when we use conceptual graphs to implement natural language understanding. Consider the three sentences:

Albert Einstein formulated the theory of relativity.

Albert Einstein plays center for the Los Angeles Lakers.

Conceptual graphs are yellow flying popsicles.

The first of these sentences is true and the second is false. The third sentence, however, is meaningless; though grammatically correct, it makes no sense. The second sentence, although false, is meaningful. I can imagine Albert Einstein on a basketball court. The canonical formation rules enforce constraints on semantic meaning; that is, they do not allow us to form nonsensical graphs from meaningful ones. Although they are not sound inference rules, canonical formation rules form a basis for much of the plausible reasoning done in natural language understanding and common sense reasoning. We discuss approach further in Section 15.5.

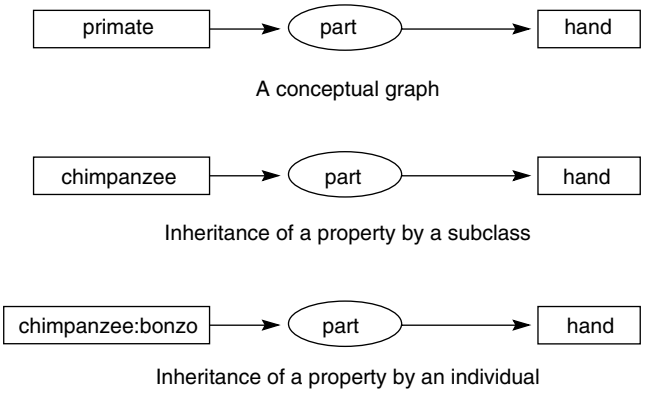


Figure 7.23 Inheritance in conceptual graphs.

7.2.5 Propositional Nodes

In addition to using graphs to define relations between objects in the world, we may also want to define relations between propositions. Consider, for example, the statement "Tom believes that Jane likes pizza". "Believes" is a relation that takes a proposition as its argument.

Conceptual graphs include a concept type, *proposition*, that takes a set of conceptual graphs as its referent and allows us to define relations involving propositions. Propositional concepts are indicated as a box that contains another conceptual graph. These proposition concepts may be used with appropriate relations to represent knowledge about propositions. Figure 7.24 shows the conceptual graph for the above assertion about Jane, Tom, and pizza. The experiencer relation is loosely analogous to the agent relation in that it links a subject and a verb. The experiencer link is used with belief states based on the notion that they are something one experiences rather than does.

Figure 7.24 shows how conceptual graphs with propositional nodes may be used to express the *modal* concepts of knowledge and belief. *Modal logics* are concerned with the various ways propositions are entertained: believed, asserted as possible, probably or necessarily true, intended as a result of an action, or counterfactual (Turner 1984).

7.2.6 Conceptual Graphs and Logic

Using conceptual graphs, we can easily represent conjunctive concepts such as "The dog is big and hungry", but we have not established any way of representing negation or disjunction. Nor have we addressed the issue of variable quantification.

We may implement negation using propositional concepts and a unary operation called neg. neg takes as argument a proposition concept and asserts that concept as false. The conceptual graph of Figure 7.25 uses neg to represent the statement "There are no pink dogs".

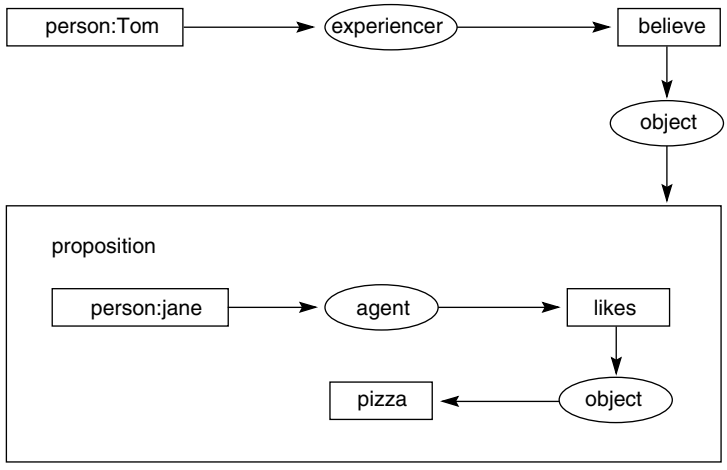


Figure 7.24 Conceptual graph of the statement Tom believes that Jane likes pizza, showing the use of a propositional concept.

Using negation and conjunction, we may form graphs that represent disjunctive assertions according to the rules of logic. To simplify this, we may also define a relation or, which takes two propositions and represents their disjunction.

In conceptual graphs, generic concepts are assumed to be existentially quantified. For example, the generic concept dog in the graph of Figure 7.14 actually represents an existentially quantified variable. This graph corresponds to the logical expression:

$$\exists X \exists Y (dog(X) \wedge color(X,Y) \wedge brown(Y)).$$

Using negation and existential quantification, Section 2.2.2, we can also represent universal quantification. For example, the graph of Figure 7.25 could be thought of as representing the logical assertion:

$$\forall X \forall Y (\neg (dog(X) \wedge color(X,Y) \wedge pink(Y))).$$

Conceptual graphs are equivalent to predicate calculus in their expressive power. As these examples suggest, there is a straightforward mapping from conceptual graphs into predicate calculus notation. The algorithm, taken from Sowa (1984), for changing a conceptual graph, g, into a predicate calculus expression is:

1. Assign a unique variable, x_1, x_2, . . . , x_n, to each of the n generic concepts in g.
2. Assign a unique constant to each individual concept in g. This constant may simply be the name or marker used to indicate the referent of the concept.
3. Represent each concept node by a unary predicate with the same name as the type of that node and whose argument is the variable or constant given that node.

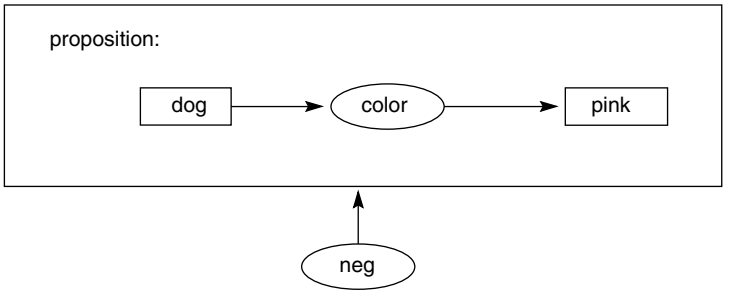


Figure 7.25 Conceptual graph of the proposition There are no pink dogs.

4. Represent each n-ary conceptual relation in g as an n-ary predicate whose name is the same as the relation. Let each argument of the predicate be the variable or constant assigned to the corresponding concept node linked to that relation.
5. Take the conjunction of all atomic sentences formed under 3 and 4. This is the body of the predicate calculus expressions. All the variables in the expression are existentially quantified.

For example, the graph of Figure 7.16 may be transformed into the predicate calculus expression

$$\exists X_1 (\text{dog}(\text{emma}) \wedge \text{color}(\text{emma}, X_1) \wedge \text{brown}(X_1))$$

Although we can reformulate conceptual graphs into predicate calculus syntax, conceptual graphs also support a number of special-purpose inferencing mechanisms, such as *join* and *restrict*, as presented in Section7.2.4, not normally part of the predicate calculus.

We have presented the syntax of conceptual graphs and defined the *restriction* operation as a means of implementing inheritance. We have not yet examined the full range of operations and inferences that may be performed on these graphs, nor have we addressed the problem of defining the concepts and relations needed for domains such as natural language. We address these issues again in Section 15.3.2 where we use conceptual graphs to implement a knowledge base for a simple natural language understanding program.

7.3 Alternative Representations and Ontologies

In recent years, AI researchers have continued to question the role of explicit representation in intelligence. Besides the connectionist and emergent approaches of Chapters 11 and 12, a further challenge to the role of traditional representation comes from Rodney Brooks' work at MIT (Brooks 1991a). Brooks questions, in designing a robot explorer, the need for *any* centralized representational scheme, and with his *subsumption architecture*, attempts to show how general intelligence might evolve from lower and supporting forms of intelligence.

A second approach to the problem of explicit and static representations comes from the work of Melanie Mitchell and Douglas Hofstadter at Indiana University. The *Copycat* architecture is an evolving network which adjusts itself to the meaning relationships that it encounters through experiment with an external world.

Finally, when building problem solvers in complex environments it is often necessary to employ multiple different representational schemes. Each representational scheme may be referred to as an *ontology*. It then becomes necessary to build communication and other links between these different representations to support knowledge management, communication, backup, and other aspects of the problem solving. We introduce these issues and possible solutions under the general heading of *knowledge management technology*. This technology can also support agent-based problem solving as presented in Section 7.4.

7.3.1 Brooks' Subsumption Architecture

Brooks conjectures, and offers examples through his robotic creations, that intelligent behavior does not come from disembodied systems like theorem provers, or even from traditional expert systems (Section 8.2). Intelligence, Brooks claims, is the product of the interaction between an appropriately designed system and its environment. Furthermore, Brooks espouses the view that intelligent behavior *emerges* from the interactions of architectures of organized simpler behaviors: his *subsumption architecture*.

The subsumption architecture is a layered collection of task-handlers. Each task is accomplished by a finite state machine that continually maps a perception-based input into an action-oriented output. This mapping is accomplished through simple sets of *condition* $\rightarrow$ *action* production rules (Section 6.2). These rules determine, in a fairly blind fashion, that is, with no global state knowledge, what actions are appropriate to the current state of that subsystem. Brooks does allow some feedback to lower level systems.

Figure 7.26, adapted from Brooks (1991a), shows a three-layered subsumption architecture. Each layer is composed of a fixed topology network of simple finite state machines, each having a few states, one or two internal registers, one or two internal timers, and access to simple computational devices, for example, to compute vector sums. These finite state machines run asynchronously, sending and receiving fixed length messages over wires. There is no central locus of control. Rather, each finite state machine is data-driven by the messages it receives. The arrival of a message or the expiration of a time period causes the machines to change state. There is no access to global data or to any dynamically created communication links. That is, there is no global control.

Figure 7.26 presents a subset of the functions of the three-layered architecture that supported an early robot (Brooks 1991a). The robot had a ring of twelve sonar sensors around it. At every second these sensors gave twelve radial depth measurements. The lowest level layer of the subsumption architecture, AVOID, implements a behavior that keeps the robot from hitting objects, whether these are static or moving. The machine labeled sonar data emits an instantaneous map that is passed on to collide and feelforce, which in turn, are able to generate halt messages for the finite state machine in charge of running the robot forward. When feelforce is activated, it is able to generate either runaway or avoid instructions for object and threat avoidance.

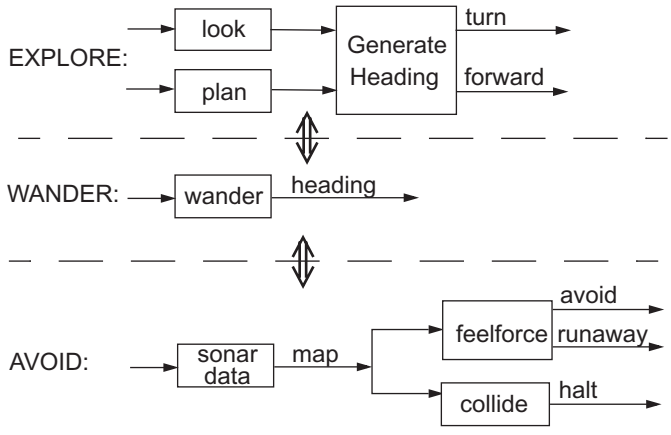


Figure 7.26 The functions of the three-layered subsumption architecture from Brooks (1991a). The layers are described by the AVOID, WANDER, and EXPLORE behaviors.

The lowest level network of finite state machines in the architecture generates all halt and avoid instructions for the entire system. The next layer, WANDER, makes the system move about by generating a random heading for the robot about every ten seconds. The AVOID (lower level) machine takes the heading from WANDER and couples it with the forces computed by the AVOID architecture. WANDER uses the result to suppress lower level behavior, forcing the robot to move in a direction close to what wander decided, but at the same time avoiding obstacles. Finally, if the turn and forward machines (top level architecture) are activated, they will suppress any new impulses sent from WANDER.

The top level, EXPLORE, makes the robot explore its environment, looking for distant places and trying to reach them by planning a path. This layer is able to suppress the wander instructions and observes how the bottom layer diverts the robot due to obstacles. It corrects for these divergences and keeps the robot focused on its goal, inhibiting the wandering behavior but allowing the lowest level object avoidance to continue its function. When deviations generated at the lower level occur, the EXPLORE level calls plan to keep the system goal-oriented. The main point of Brook's subsumption architecture is that the system requires no centralized symbolic reasoning and takes all its actions without searching through possible next states. Although the behavior-based finite state machine is generating suggestions for actions based on its own current state, the global system acts based on the interactions of the systems layered below it.

The three-layer architecture just presented comes from Brooks' early design of a wandering, goal-seeking robot. More recently, his research group has built complex systems

with further layers (Brooks 1991a, Brooks and Stein 1994). One system is able to wander around the MIT Robotics Lab looking for empty aluminum drink cans on people's desks. This requires layers for discovering offices, looking for desks, and recognizing drink cans. Further layers are able to guide the robot arm to collect these cans for recycling.

Brooks insists that top level behavior emerges as a result of the design and testing of the individual lower level layers of the architecture. The design for coherent final behaviors, requiring both inter-layer and between-layer communications, is discovered through experiment. Despite this simplicity of design, the subsumption architecture has performed successfully in several applications (Brooks 1989, 1991a, 1997).

There remains, a number of important questions concerning the subsumption architecture and related approaches to the design of control systems (see also Section 16.2):

1. There is a problem of the sufficiency of information at each system level. Since at each level, purely reactive state machines make decisions on local information, it is difficult to see how such decisions can take account of information not at that level. By definition, it will be myopic.

2. If there exists absolutely no "knowledge" or "model" of the complete environment, how can the limited input on the local situation be sufficient for determination of globally acceptable actions? How can top level coherence result?

3. How can a purely reactive component with very limited state *learn* from its environment? At some level in the system, there must be sufficient state for the creation of learning mechanisms if the overall agent is to be called intelligent.

4. There is a problem of scale. Although Brooks and his associates claim to have built subsumption architectures of six and even ten layers, what design principles allow it to scale to further interesting behavior, i.e., to large complex systems?

Finally, we must ask the question what is "emergence"? Is it magic? At this time in the evolution of science, "emergence" seems to be a word to cover any phenomenon for which we can, as yet, make no other accounting. We are told to build systems and test them in the world and they will show intelligence. Unfortunately, without further design instruction, emergence just becomes a word to describe what we can't yet understand. As a result, it is very difficult to determine how we can *use* this technology to build ever more complex systems. In the next section we describe *copycat*, a hybrid architecture that is able by exploration to discover and use invariances found in a problem domain.

7.3.2 Copycat

An often-heard criticism of traditional AI representation schemes is that they are static and cannot possibly reflect the dynamic nature of thought processes and intelligence. When a human perceives a new situation, for example, he or she is often struck by relationships with already known or analogous situations. In fact, it is often noted that human percep-

tion is both bottom up, that is, stimulated by new patterns in the environment, and top down, mediated by what the agent expects to perceive.

Copycat is a problem-solving architecture built by Melanie Mitchell (1993) as a PhD dissertation under Douglas Hofstadter (1995) at Indiana University. Copycat builds on many of the representational techniques that preceded it, including blackboards, Section 6.3, semantic networks, Section 7.2, connectionist networks, Chapter 11, and classifier systems (Holland 1986). It also follows Brooks' approach to problem solving as active intervention in a problem domain. In contrast with Brooks and connectionist networks, however, copycat requires a global "state" to be part of the problem solver. Secondly, representation is an evolving feature of that state. Copycat supports a semantic network-like mechanism that grows and changes with continuing experience within its environment.

The original problem domain for copycat was the perception and building of simple analogies. In that sense it is building on earlier work by Evans (1968) and Reitman (1965). Examples of this domain are completing the patterns: hot is to cold as tall is to {wall, short, wet, hold} or bear is to pig as chair is to {foot, table, coffee, strawberry}. Copycat also worked to discover appropriate completions for alphabetic string patterns such as: abc is to abd as ijk is to ? or again, abc is to abd as iijjkk is to ?, Figure 7.27.

Copycat is made up of three components, the *workspace*, the *slipnet*, and the *coderack*. These three are mediated in their interactions by a *temperature* measure. The temperature captures the degree of perceptual organization in the system, and controls the degree of randomness used in making decisions. Higher temperatures reflect the fact that there is little information on which to base decisions and thus they are more random. A temperature drop indicates the system is building consensus and a low temperature indicates that an answer is emerging and reflects the program's "confidence" in that solution.

The *workspace* is a global structure, similar to the blackboard of Section 6.3, for creating structures that the other components of the system can inspect. In this sense it is also much like the message area in Holland's (1986) classifier system. The workspace is where perceptual structures are built hierarchically on top of the input (the three strings of alphabetic symbols of Figure 7.27) and gives possible states for the workspace, with bonds (the arrows) built between related components of the strings.

The *slipnet* reflects the network of concepts or potential associations for the components of the analogy. One view of the slipnet is as a dynamically deformable semantic network, each of whose nodes has an activation level. Links in the network can be labeled by other nodes. Based on the activation level of the labelling nodes, the linked nodes grow or shrink. In this way the system changes the degree of association between the nodes as a function of context. The spreading activation among nodes is encouraged between nodes that (in the current context) are more closely related.

The *coderack* is a priority based probabilistic queue containing *codelets*. Codelets are small pieces of executable code designed to interact with the objects in the workspace and to attempt to further some small part of the evolving solution, or, more simply, to explore different facets of the problem space. Again, the codelets are very much like the individual classifiers of Holland's (1986) system.

Nodes in the slipnet generate codelets and post them to the coderack where they have a probabilistic chance of execution. The system maintains pieces of code in parallel that compete for the chance to find and build structures in the workspace. These pieces of code

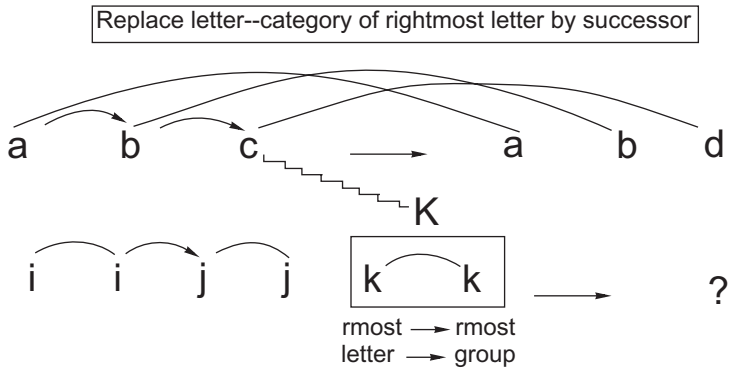


Figure 7.27 A possible state of the copycat workspace. Several examples of bonds and links between the letters are shown; adopted from Mitchell (1993).

correspond to the nodes from which they came. This is a top-down activity, seeking more examples of the things that have already generated interest. Codelets also work bottom up, identifying and building on relationships already existing in the workspace.

As structures are built in the workspace, an *activation* measure is added to the nodes that generated the codelets that built the structures. This allows the system's behavior in a context of the current state of the problem/solution to impact its future behavior.

Finally, the *temperature* serves as a feedback mechanism calculating the "cohesion" among the structures built in the workspace. With little cohesion, that is, few structures that offer promise of a solution, the biases of the probabilistic choices made by the system are less important: one choice is just about as useful as another. When the cohesion is high with an internally consistent solution evolving, biases become very important in the probabilistic choices: the choices made are more closely linked to the evolving solution.

There are several follow-on projects in the copycat world that test the architecture in richer settings. Hofstadter and his students (Marshall 1999) continue to model analogical relationships. Lewis and Luger (2000) have expanded the copycat architecture for use as a control system for a mobile robot. The robotics domain gives copycat a concrete environment which supports learning-oriented interactions. Evolving from the robot's interaction with its "world" a map of the space is generated and paths are planned and replanned.

7.3.3 Multiple Representations, Ontologies, and Knowledge Services

Many complex AI applications require that multiple representational schemes be designed, each set up for specific casks, and then integrated into a user oriented software platform. A simple example of this might be the creation of a virtual meeting space. The various attendees, from their remote locations would interact with each other in this virtual

"space". Besides voice technology, the virtual meeting space would require video streaming, text and presentation sharing, records access, and so on. One approach to this task would be to adapt various off-the-shelf components and then integrate them into a knowledge service that supports the interactions required.

The automated virtual meeting service just described is an example of integrating multiple representational schemes to provide a service. We often use the term *ontology* to characterize the representation scheme of a particular component of that service. The resulting knowledge service requires the integration of these components. Ontology is derived from a Greek word, a form of the verb "to be", that means *being* or *existence*. Thus the term refers to the components of a software module and the structure supporting the interactions of these components, that is, to their function or "constituent being" as part of the module.

When the modules are composed to build the knowledge service each of the modules have to be appropriately linked to form the seamless service. We say that the ontologies of each module are merged to make a new more complex ontology that supports the full service. To create a new knowledge service, such as software to support the virtual meeting, the ontologies of each of its components must be understood and often exposed for the integration. There now exists a number of ontology languages designed explicitly for the creation of knowledge services and other related software products. These include the Owl family of languages (Mika et al. 2004) as well as the open source Java api SOFA, Simple Ontology Framework, (see http://sopha.projects.semwebcentral.org/).

There are if fact multiple ways that ontological information is used in modern computing, some straightforward, such as the optimized access to knowledge stored in a traditional database, and others quite novel. With the importance of the WWW, sophisticated web crawlers and search engines are critical support tools. Although most current web spiders apply their search to flat directly linked text files, a more sophisticated crawler can be envisioned that can traverse the classes and links of particular complex representations as well as "interpret" more useful entities including graphs and images.

Although full "image interpretation" remains well beyond the current state of our capabilities, its creation will help support our desires for a more "semantic" web. We may want to scan our holiday photographs to find all that have a particular person in them, or any animal, for example. In the larger situation of the WWW we may want to not only track and find any newspaper stories that describe natural disasters but also locate any images that present these disasters. Currently we are often limited to the search a figure's captions or other (textual) descriptive materials for a hint of their content.

Besides the goal of interpreting graphical and image data, it is also desired to locate text data on specific topics and produce a summary of its content. For instance, we might want to search for on line research articles in a certain area and report back their contents. Although this is another current research topic in the area of natural language understanding, Chapter 15, its implementation remains at the boundary of our computer based "semantic" skills. Currently we do things such as searching for a few key words that might be indicative of an article's content and perhaps print out its abstract, but other than this our summarization skills are limited (Ciravegna and Wilks 2004, Wilks 2004).

A further web service might be desired to search for "help wanted" advertisement. It might be launched to search for software professionals, for example, with skills in Java

and C++ and willing to locate in a specific city. This web crawler would need to know something about this type advertisement, including the facts that the company name and its location are usually included. Besides the skills required for the job, these ads often describe experience needed along with possible salaries and benefits. Thus the employment web service must have knowledge of and the ability to accesses the ontological structures used in the design of employment "want ads". We present an example of such a knowledge service in Section 15.5.

Perhaps the most important application, however, for ontology based knowledge services is in agent-oriented problem solving. Agents by definition are intended to be autonomous, independent, flexible when addressing tasks, located in specific situations, and able to communicate appropriately. Agent-based problem solving is also highly distributed across environments, such as might be required for constructing an intelligent web service. The creation of specific agent skills can be seen as a question for building and linking ontologies. In the next section on agent-based problem solving, we consider further the distributed and component-oriented view of representation.

7.4 Agent-Based and Distributed Problem Solving

There were two insights in the AI research community in the 1980s that had important consequences for future research in the analysis of the role of representation in intelligent problem solving. The first was the research tradition of the "Distributed Artificial Intelligence" or the "DAI" community. The first DAI workshop was held at MIT in 1980 for discussion of issues related to intelligent problem solving with systems consisting of multiple problem solvers. It was decided at that time by the DAI community that they were *not* interested in low level parallelism issues, such as how to distribute processing over different machines or how to parallelize complex algorithms. Rather, their goal was to understand how distributed problem solvers could be effectively coordinated for the intelligent solution of problems. In fact, there had been an even earlier history of distributed processing in artificial intelligence, with the use and coordination of *actors* and *demons* and the design of blackboard systems, as seen in Section 6.3.

The second AI research insight of the 1980s, already presented in Section 7.3, was that of Rodney Brooks and his group at MIT. Brooks' challenge to the AI community's traditional view of representation and reasoning had many important consequences (Section 7.3.1). First, the conjecture that intelligent problem solving does not require a centralized store of knowledge manipulated by some general-purpose inferencing scheme led to the notion of distributed and cooperative models of intelligence, where each element of the distributed representation was responsible for its own component of the problem solving process. Second, the fact that intelligence is situated and active in the context of particular tasks, allows the problem solver to offload aspects of the solution process into the environment itself. This allows, for example, an individual solver to address a task at hand and at the same time have no knowledge whatsoever of the progress towards solution within the general problem domain. Thus one web agent, say, can check inventory information, while another agent checks credit worthiness of a customer, both agents unaware of the higher level decision making, for example, whether or not to allow a purchase. Both

of these research emphases of the 1980s brought on the current interest in *the design and use of intelligent agents*.

7.4.1 Agent-Oriented Problem Solving: A Definition

Before proceeding further in the discussion of agent research, we define what we mean by "agent", "agent-based system", and "multi-agent system". There are problems here, however, as many different groups in the agent research community have differing opinions as to exactly what agent-based problem solving is about. Our definition and discussion is based on an extension of the work by Jennings, Sycara, and Wooldridge (1998), Wooldridge (2000), Wooldridge et al. (2006, 2007), and Lewis and Luger (2000).

For us, a multi-agent system is a computer program with problem solvers situated in interactive environments, which are each capable of flexible, autonomous, yet socially organized actions that can, but need not be, directed towards predetermined objectives or goals. Thus, the four criteria for an intelligent agent system include software problem solvers that are *situated*, *autonomous*, *flexible*, and *social*.

That an intelligent agent is *situated* means that the agent receives input from the environment in which it is active and can also effect changes within that environment. Examples of environments for situated agents include the internet, game playing, or a robotics application. A concrete example might be a soccer player in a ROBOCUP competition (Veloso et al. 2000) where an agent must interact appropriately with the ball and an opponent without full knowledge of the locations, challenges, and successes of other players in the contest. This situatedness can be contrasted with more traditional AI problem solvers, such as the STRIPS planner, Section 8.4, or the MYCIN expert system, Section 8.3, that maintain centrally located and exhaustive knowledge of application domains.

An *autonomous* system is one that can interact with its environment without the direct intervention of other agents. To do this it must have control over its own actions and internal state. Some autonomous agents can also learn from their experience to improve their performance over time (see machine learning, Part IV). For example, on the internet, an autonomous agent could do a credit card authentication check independent of other issues in the purchasing transaction. In the ROBOCUP example, an agent could pass the ball to a teammate or kick it on goal depending on its individual situation.

A *flexible* agent is both intelligently *responsive* as well as *proactive* depending on its current situation. A responsive agent receives stimuli from its environment and responds to them in an appropriate and timely fashion. A proactive agent does not simply respond to situations in its environment but is also able to plan, be opportunistic, goal directed, and have appropriate alternatives for various situations. A credit agent, for example, would be able to go back to the user with ambiguous results or find another credit agency if one alternative is not sufficient. The soccer agent could change its trajectory depending on the challenge pattern of an opponent.

Finally, an agent is *social* that can interact, as appropriate, with other software or human agents. After all, an agent is only part of a complex problem solving process. The interactions of the social agent are oriented towards the goals of the larger multi-agent system. This social dimension of the agent system must address many difficult situations.

These include: How can different agents bid for a subtask in problem solving? How can agents communicate with each other to facilitate the accomplishment of higher system-level tasks - in the ROBOCUP example, this might be to score a goal. How can one agent support another agent's goals, for example, to handle the security issues of an internet task? All these questions on the social dimension are the subject of ongoing research.

We have described the basis for creating multi-agent systems. Multi-agent systems are ideal for representing problems that include many problem-solving methods, multiple viewpoints, and multiple entities. In these domains, multi-agent systems offer the advantages of distributed and concurrent problem solving along with the advantages of sophisticated schemes for interaction. Examples of interactions include cooperation in working towards a common goal, coordination in organizing problem-solving activity so that harmful interactions are avoided and beneficial possibilities exploited, and negotiation of subproblem constraints so that acceptable performance ensues. It is the flexibility of these social interactions that distinguishes multi-agent systems from more traditional software and which provides the power and excitement to the agent paradigm.

In recent years, the term *multi-agent system* refers to all types of software systems composed of multiple semi-autonomous components. The distributed agent system considers how a particular problem can be solved by a number of modules (agents) which cooperate by dividing and sharing the knowledge about the problem and its evolving solution. Research in multi-agent systems is focused on the behaviors of collections of, sometimes already existing, autonomous agents aimed at solving a given problem. A multi-agent system can also be seen as a loosely coupled network of problem solvers that work together on problems beyond the scope of any of the agents individually (Durfee and Lesser 1989). See also the design of knowledge services and ontologies, Section 7.3.3.

The problem solvers of a multi-agent system, besides being autonomous, may also be of heterogeneous design. Based on analysis by Jennings, Sycara, and Wooldridge (1998), Wooldridge et al. (2006, 2007) there are four important characteristics of multi-agent problem solving. First, each agent has incomplete information and insufficient capabilities for solving the entire problem, and thus can suffer from a limited viewpoint. Second, there is no global system controller for the entire problem solving. Third, the knowledge and input data for the problem is also decentralized, and fourth, the reasoning processes are often asynchronous.

Interestingly, traditional object-oriented programmers often fail to see anything new in an agent-based system. On consideration of the relative properties of agents and objects, this can be understandable. Objects are defined as computational systems with encapsulated state, they have methods associated with this state that support interactions in an environment, and they communicate by message passing.

Differences between objects and agents include the fact that objects rarely exhibit control over their own behavior. We do not see agents as invoking methods on one another, but rather as requesting that service actions be performed. Further, agents are designed to have flexible, i.e., reactive, proactive, and social behavior. Finally, interacting agents are often have their own individual threads of control. All these differences are not to indicate that object-oriented programming languages, such as C++, Java, or CLOS do not offer a suitable medium for building agent systems; quite to the contrary, their power and flexibility make them ideal for this task.

7.4.2 Examples of and Challenges to an Agent-Oriented Paradigm

To make the ideas of the previous section more concrete we next describe a number of application domains where agent-based problem solving is appropriate. We also include references to research within these problem areas.

Manufacturing The manufacturing domain can be modeled as a hierarchy of work areas. There may be work areas for milling, lathing, painting, assembling, and so on. These work areas may then be grouped into manufacturing subsystems, each subsystem a function within the larger manufacturing process. These manufacturing subsystems may then be grouped into a single factory. A larger entity, the company, may then control aspects of each of these factories, for instance, to manage orders, inventory, levels of duplication, profits, and so on. References for agent-based manufacturing include work in production sequencing (Chung and Wu 1997), manufacturing operations (Oliveira et al. 1997), and the collaborative design of products (Cutosky et al. 1993, Darr and Birmingham 1996, and Woldridge et al. 2007).

Automated Control Since process controllers are autonomous, reactive, and often distributed systems, it is not surprising that agent models can be important. There is research in controlling transportation systems (Corera et al. 1996), spacecraft control (Schwuttke and Quan 1993), particle beam accelerators (Perriolat et al. 1996, Klein et al. 2000), air traffic control (Ljunberg and Lucas 1992) and others.

Telecommunications Telecommunication systems are large distributed networks of interacting components that require real-time monitoring and management. Agent-based systems have been used for network control and management (Schoonderwoerd et al. 1997, Adler et al. 1989, Fatima et al. 2006), transmission and switching (Nishibe et al. 1993), and service (Busuoic and Griffiths 1994). See (Veloso et al. 2000) for a comprehensive overview.

Transportation Systems Traffic systems are almost by definition distributed, situated, and autonomous. Applications include coordinating commuters and cars for carpooling (Burmeister et al. 1997) and cooperative transportation scheduling (Fischer et al. 1996).

Information Management The richness, diversity, and complexity of information available to current society is almost overwhelming. Agent systems allow the possibility of intelligent information management, especially on the internet. Both human factors as well as information organization seem to conspire against comfortable access to information. Two critical agent tasks are information filtering, only a tiny portion of the information that we have access to do we actually want, and information gathering, the task of collecting and prioritizing the pieces of information that we actually do want. Applications include WEBMATE (Chen and Sycara 1998), electronic mail filtering (Maes 1994), a web browsing assistant (Lieberman 1995), and an expert locator agent (Kautz et al. 1997). See also knowledge services, Section 7.3.3.

E-Commerce Commerce currently seems to be driven by human activity: we decide when to buy or sell, the appropriate quantities and price, even what information might be appropriate at the time. Certainly commerce is a domain appropriate for agent models. Although full development of e-commerce agents may be in the future, several systems are already available. For example, programs can now make many buy and sell decisions in the stock market, based on many diverse pieces of distributed information. Several agent systems are being developed for portfolio management (Sycara et al. 1996), shopping assistance (Doorenbos et al. 1997, Krulwich 1996), and interactive catalogues (Schrooten and van de Velde 1997, Takahashi et al. 1997).

Interactive Games and Theater Game and theater characters provide a rich interactive simulated environment. These agents can challenge us in war games, finance management scenarios, or even sports. Theater agents play roles analogous to their human counterparts and can offer the illusion of life for working with emotional situations, simulating medical emergencies, or training for diverse tasks. Research in this area includes computer games (Wavish and Graham 1996), interactive personalities (Hayes-Roth 1995, Trappl and Petta 1997), and negotiation (Fatima et al. 2006).

There are many other domains, of course, where agent-based approaches are appropriate.

Even though the agent technology offers many potential advantages for intelligent problem solving it still has a number of challenges to overcome. The following research questions are based on ideas from Jennings et al. (1998) and Bond and Gasser (1988).

How can we systematically formalize, decompose, and allocate problems to agents? Furthermore, how do we appropriately synthesize their results?

How can we enable agents to communicate and interact? What communication languages and protocols are available? What and when is communication appropriate?

How can we ensure that agents act coherently in taking actions or making decisions? How can they address nonlocal effects and avoid harmful agent interactions?

How can individual agents represent and reason about the actions, plans, and knowledge of other agents in order to coordinate with them? How can agents reason about the state of their coordinated processes?

How can disparate viewpoints and conflicting intentions between agents be recognized and coordinated?

How can harmful overall system behavior, such as chaotic or oscillatory action, be recognized and avoided?

How can limited resources, both of the individual agent as well as for the full system, be allocated and managed?

Finally, what are the best hardware platforms and software technologies for the support and development of agent systems?

The intelligent software design skills necessary to support agent problem-solving technology are found throughout this book. First, the representational requirements for intelligent problem solving make up a constant theme of our presentation. Second, issues of search, especially heuristic search, may be found in Part II. Third, the area of *planning*, presented in Section 8.4, provides a methodology of ordering and coordinating subgoals in the process of organizing a problem solution. Fourth, we present the idea of stochastic agents reasoning under conditions of uncertainty in Section 9.3. Finally, issues in learning, automated reasoning, and natural language understanding are addressed in Parts V and IV. These subareas of traditional AI have their roles within the creation of agent architectures.

There are a number of other design issues for the agent model that go beyond the scope of this book, for example agent communication languages, bidding schemes, and techniques for distributed control. These are addressed in the agent literature (Jennings, 1995; Jennings et al. 1998; Wooldridge 1998; Wooldridge et al. 2006, 2007; Fatima et al. 2005, 2006), and in particular in the appropriate conference proceedings (AAAI, IJCAI, and DAI).

7.5 Epilogue and References

In this chapter, we have examined many of the major alternatives for knowledge representation, including the use of logic, rules, semantic networks, and frames. We also considered systems with no centralized knowledge base or general-purpose reasoning scheme. Finally, we considered distributed problem-solving with agents. The results of careful study include an increased understanding of the advantages and limitations of each of these approaches to representation. Nonetheless, debate continues over the relative naturalness, efficiency, and appropriateness of each approach. We close this chapter with a brief discussion of several important issues in the area of knowledge representation.

The first of these is the *selection and granularity of atomic symbols* for representing knowledge. Objects in the world constitute the domain of the mapping; computational objects in the knowledge base are the range. The nature of the atomic elements in the language largely determines what can be described about the world. For example, if a "car" is the smallest atom of the representation, then the system cannot reason about engines, wheels, or any of the component parts of a car. However, if the atoms correspond to these parts, then a larger structure may be required to represent "car" as a single concept, possibly introducing a cost in efficiency in manipulating this larger structure.

Another example of the trade-off in the choice of symbols comes from work in natural language understanding. Programs that use single words as elements of meaning may have difficulty in representing complex concepts that do not have a one-word denotation. There is also difficulty in distinguishing between different meanings of the same word or different words with the same meaning. One approach to this problem is to use semantic primitives or language-independent conceptual units, as the basis for representing the meaning of language. Although this viewpoint avoids the problem of using single words as units of meaning, it involves other trade-offs: many words require complex structures for their definitions; also, by relying on a small set of primitives, many subtle distinctions, such as push vs. shove or yell vs. scream, are difficult to express.

Exhaustiveness is a property of a knowledge base that is assisted by an appropriate representation. A mapping is *exhaustive* with respect to a property or class of objects if all occurrences correspond to an explicit element of the representation. Geographic maps are assumed to be exhaustive to some level of detail; a map with a missing city or river would not be well regarded as a navigational tool. Although most knowledge bases are not exhaustive, exhaustiveness with respect to certain properties or objects is a desirable goal. For example, the ability to assume that a representation is exhaustive may allow a planner to ignore possible effects of the *frame problem*.

When we describe problems as a state of the world that is changed by a series of actions or events, these actions or events generally change only a few components of the description; the program must be able to infer side effects and implicit changes in the world description. The problem of representing the side effects of actions is called the *frame problem*. For example, a robot stacking heavy boxes on a truck may have to compensate for the lowering of the truck bed due to the weight of each new box. If a representation is exhaustive, there will be no unspecified side effects, and the frame problem effectively disappears. The difficulty of the frame problem results from the fact that it is impossible to build a completely exhaustive knowledge base for most domains. A representation language should assist the programmer in deciding what knowledge may safely be omitted and help deal with the consequences of this omission. (Section 8.4 discusses the frame problem in planning.)

Related to exhaustiveness is the *plasticity* or modifiability of the representation: the addition of knowledge in response to deficiencies is the primary solution to a lack of exhaustiveness. Because most knowledge bases are not exhaustive, it should be easy to modify or update them. In addition to the syntactic ease of adding knowledge, a representation should help to guarantee the consistency of a knowledge base as information is added or deleted. Inheritance, by allowing properties of a class to be inherited by new instances, is an example of how a representational scheme may help ensure consistency.

Several systems, including Copycat (Mitchell 1993) and Brooks' robot, Section 7.3, have addressed the plasticity issue by designing network structures that change and evolve as they meet the constraints of the natural world. In these systems, the representation is the result of bottom-up acquisition of new data, constrained by the expectation of the perceiving system. An application of this type of system is analogical reasoning.

Another useful property of representations concerns the extent to which the mapping between the world and the knowledge base is *homomorphic*. Here, homomorphic implies a one-to-one correspondence between objects and actions in the world and the computational objects and operations of the language. In a homomorphic mapping the knowledge base reflects the perceived organization of the domain and can be organized in a more natural and intuitive fashion.

In addition to naturalness, directness, and ease of use, representational schemes may also be evaluated by their *computational efficiency*. Levesque and Brachman (1985) discuss the trade-off between expressiveness and efficiency. Logic, when used as a representational scheme, is highly expressive as a result of its completeness; however, systems based on unconstrained logic pay a considerable price in efficiency, see Chapter 14.

Most of the representation issues just presented relate to any information that is to be captured and used by a computer. There are further issues that the designers of distributed

and agent systems must address. Many of these issues relate to making decisions with partial (local) information, distributing the responsibility for accomplishing tasks, agent communication languages, and developing algorithms for the cooperation and information sharing of agents. Many of these issues are presented in Section 7.4.

Finally, if the philosophical approaches of a distributed environment-based intelligence proposed by Clark (1997), Haugeland (1997), and Dennett (1991, 1995, 2006) are to be realized, where the so-called "leaky" system utilizes both the environment and other agents as critical media for knowledge storage and use, it may be that entirely new representation languages await invention. Where are the representational tools and support to, as Brooks (1991a) proposes, "use the world as its own model"?

We conclude with further references for the material presented in Chapter 7. Associationist theories have been studied as models of computer and human memory and reasoning (Selz 1913, 1922; Anderson and Bower 1973; Sowa 1984; Collins and Quillian 1969).

SNePS, (Shapiro 1979, Shapiro et al. 2007), one of the earliest associationist/logic based representations, see Section 7.1, has been recently extended as a tool for metacognition, where predicates are allowed to take other predicates as arguments, thus supporting the ability to reason about the structures of a knowledge base.

Important work in structured knowledge representation languages includes Bobrow and Winograd's representation language KRL (Bobrow and Winograd 1977) and Brachman's (1979) representation language KL-ONE, which pays particular attention to the semantic foundations of structured representations.

Our overview of conceptual graphs owes a considerable debt to John Sowa's book *Conceptual Structures* (1984). The reader is referred to this book for details that we have omitted. In its full treatment, conceptual graphs combine the expressive power of predicate calculus, as well as that of modal and higher-order logics, with a rich set of built-in concepts and relations derived from epistemology, psychology, and linguistics.

There are a number of other approaches of interest to the representation problem. For example, Brachman, Fikes, and Levesque have proposed a representation that emphasizes *functional* specifications; that is, what information can be asked of or told to a knowledge base (Brachman 1985, Brachman et al. 1985, Levesque 1984).

A number of books can help with an advanced study of these issues. *Readings in Knowledge Representation* by Brachman and Levesque (1985) is a compilation of important, but dated articles in this area. Many of the articles referred to in this chapter may be found there, although we have referenced them in their original source. *Representation and Understanding* by Bobrow and Collins (1975), *Representations of Commonsense Knowledge* by Davis (1990), *Readings in Qualitative Reasoning about Physical Systems* by Weld and deKleer (1990) are all important. *Principles of Knowledge Representation and Reasoning,* (Brachman et al. 1990), *An Overview of Knowledge Representation* (Mylopoulos and Levesque 1984), and the proceedings of the annual conferences on AI are helpful resources.

There is now a considerable number of papers following the directions proposed by Brooks with his subsumption architecture; see especially Brooks (1991a), Brooks and Stein (1994), Maes (1994), and Veloso et al. (2000). There are also contrasting positions, see McGonigle (1990, 1998) and Lewis and Luger (2000). For a philosophical view of distributed and embodied knowledge and intelligence see Clark's *Being There* (1997).

Agent-based research is widespread in modern AI. See Jennings et al. (1998) for an introduction to the area. There are now entire sections of the annual AI conferences (IAAI and IJCAI) devoted to agent research. We recommend reading the recent proceedings of these conferences for more up-to-date discussions of current research issues. We also recommend the conference proceedings and readings in the area of distributed artificial intelligence, or DAI. A summary and references of important application areas for agent-based research was presented in Section 7.4.

Issues in knowledge representation also lie in the middle ground between AI and Cognitive Science; see *Cognitive Science: The Science of Intelligent Systems* (Luger 1994) and *Being There* (Clark 1997). *Computation and Intelligence* edited by George Luger (1995) is a collection of classic papers that emphasizes the development of many different knowledge representation schemes.

7.6 Exercises

1. Common sense reasoning employs such notions as causality, analogy, and equivalence but uses them in a different way than do formal languages. For example, if we say "Inflation caused Jane to ask for a raise," we are suggesting a more complicated causal relationship than that found in simple physical laws. If we say "Use a knife or chisel to trim the wood," we are suggesting an important notion of equivalence. Discuss the problems of translating these and other such concepts into a formal language.
2. In Section 7.2.1 we presented some of the arguments against the use of logic for representing common sense knowledge. Make an argument for the use of logic in representing this knowledge. McCarthy and Hayes (1969) have an interesting discussion on this.
3. Translate each of the following sentences into predicate calculus, conceptual dependencies, and conceptual graphs:

   "Jane gave Tom an ice cream cone."
   "Basketball players are tall."
   "Paul cut down the tree with an axe."
   "Place all the ingredients in a bowl and mix thoroughly."
4. Read "What's in a Link" by Woods (1985). Section IV of this article lists a number of problems in knowledge representation. Suggest a solution to each of these problems using logic, conceptual graphs, and frame notations.
5. Translate the conceptual graphs of Figure 7.28 into English sentences
6. The operations *join* and *restrict* define a generalization ordering on conceptual graphs. Show that the generalization relation is transitive.
7. Specialization of conceptual graphs using *join* and *restrict* is not a truth-preserving operation. Give an example that demonstrates that the restriction of a true graph is not necessarily true. However, the generalization of a true graph is always true; prove this.
8. Define a specialized representation language to describe the activities of a public library. This language will be a set of concepts and relations using conceptual graphs. Do the same thing for a retail business. What concepts and relations would these two languages have in common? Which would exist in both languages but have a different meaning?

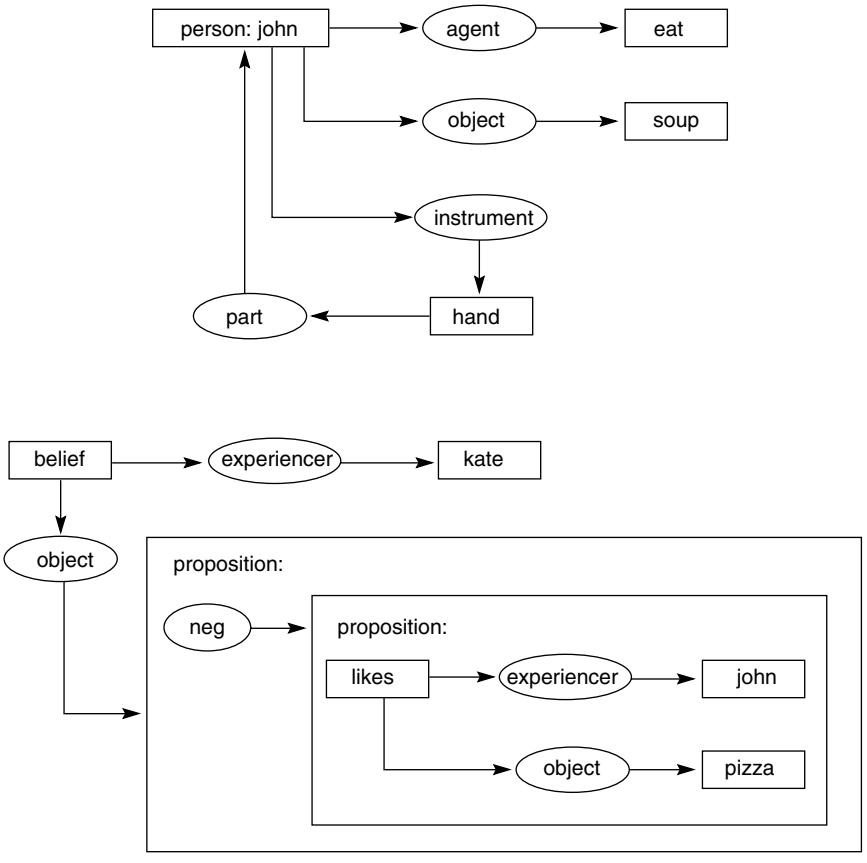


Figure 7.28 Two conceptual graphs to be translated into English.

9. Translate the conceptual graphs of Figure 7.28 into predicate calculus.
10. Translate the financial advisor knowledge base, Section 2.4, into conceptual graph form.
11. Give evidence from your own experience that suggests a script-like or frame-like organization of human memory.
12. Using conceptual dependencies, define a script for:
    a. A fast-food restaurant.
    b. Interacting with a used-car salesperson.
    c. Going to the opera.
13. Construct a hierarchy of subtypes for the concept vehicle; for example, subtypes of vehicle might be land_vehicle or ocean-vehicle. These would have further subtypes. Is this best represented as a tree, lattice, or general graph? Do the same for the concept move; for the concept angry.

14. Construct a type hierarchy in which some types do not have a common supertype. Add types to make this a lattice. Could this hierarchy be expressed using tree inheritance? What problems would arise in doing so?

15. Each of the following sequences of characters is generated according to some general rule. Describe a representation that could be used to represent the rules or relationships required to continue each sequence:

    a. 2,4,6,8, . . .
    b. 1,2,4,8,16, . . .
    c. 1,1,2,3,5,8, . . .
    d. 1,a,2,c,3,f,4, . . .
    e. o,t,t,f,f,s,s, . . .

16. Examples of analogical reasoning were presented in Section 7.3.2. Describe an appropriate representation and search strategy that would allow for identification of the best answer for this type problem. Create two more example analogies that would work with your proposed representation. Could the two following examples be solved?

    a. hot is to cold as tall is to {wall, short, wet, hold}
    b. bear is to pig as chair is to {foot, table, coffee, strawberry}

17. Describe a representation that could be used in a program to solve analogy problems like that in Figure 7.29. This class of problems was addressed by T. G. Evans (1968). The representation must be capable of capturing the essential features of size, shape, and relative position.

18. Brooks' paper (1991a) offers an important discussion on the role of representation in traditional AI. Read this paper, and comment on the limitations of explicit, general-purpose representational schemes.

19. At the end of Section 7.3.1, there are five potential issues that Brooks' subsumption architecture (1991a) must address to offer a successful general-purpose approach to problem-solving. Pick one or more of these and comment on it (them).

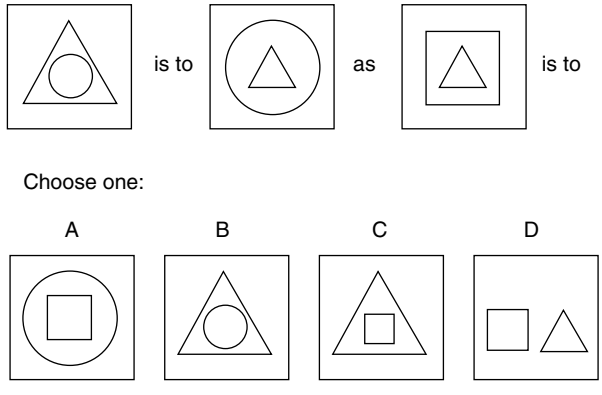


Figure 7.29 Example of an analogy test problem.

20. Identify five properties that an agent language should have to provide an agent-oriented internet service. Comment on the role of Java as a general-purpose agent language for building internet services. Do you see a similar role for CLOS? Why or why not? There is plenty of information on this topic on the internet itself.

21. There were a number of important issues presented near the end of Section 7.4 related to the creation of an agent-oriented solutions to problems. Pick one of these and discuss the issue further.

22. Suppose you were designing an agent system to represent an American football or alternatively a soccer team. For agents to cooperate in a defensive or in a scoring maneuver, they must have some idea of each other's plans and possible responses to situations. How might you build a model of another cooperating agent's goals and plans?

23 Pick one of the application areas for agent architectures summarized in Section 7.4. Choose a research or application paper in that area. Design an organization of agents that could address the problem. Break the problem down to specify issues of responsibility for each agent. List appropriate cooperation procedures.