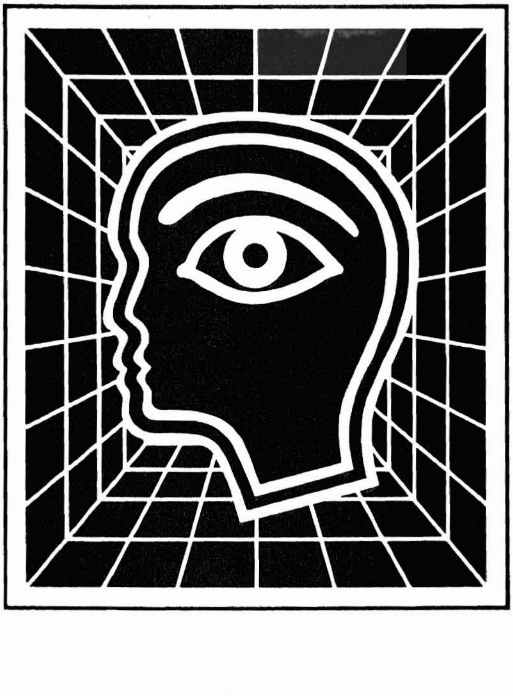
Part Two

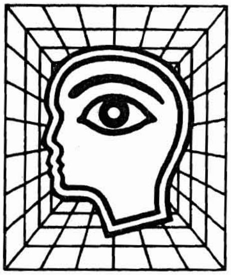
Cognition



In this part of the book we deal with the general symbolic machinery that provides a basis for reasoning, planning, and communication. A main theme is the need for "representing" a problem in a form that permits its effective solution, and the difficulty of obtaining such representations automatically.

4

Reasoning and Problem Solving



In its most basic sense, reasoning is the ability to solve problems. However, simply because a device can solve a problem does not mean that it is capable of reasoning. For example, a pocket calculator can "solve" a variety of mathematical problems, but certainly such problem solving ability is not an example of reasoning. What are some of the necessary conditions that distinguish reasoning from "mechanical" behavior?

First, we require that a reasoning system be capable of expressing and solving a broad range of problems and problem types—including problem formulations that do not correspond to rigid templates anticipated by the system designer. (The pocket calculator, for example, fails this test.) Thus, our first requirement implies that a reasoning system be based on a set of representations that has broad expressive power.

Second, the system must be able to make *explicit* the *implicit* information that is known to it, i.e., for any information possessed by the system it can systematically obtain all equivalent representations of this information.[1] For example, from the information (1) all tigers are dangerous, and (2) this animal is a tiger, the system should be able to obtain the explicit statement (3) this animal is dangerous.

[1] This is meant in a conceptual sense; in practice, obtaining all equivalent representations could take an impractical amount of time.

Note that (3) was implicit in statements (1) and (2). Thus, we require a set of operations or transformations that produce other "valid" representations when applied to the representations of information possessed by the system. The system must be able to translate a problem situation expressed in some external representation into its own (internal)representation as well as being able to "syntactically" transform information already expressed in its own formalism. For example, a translation may be made from natural language to a logic formalism, and expressions in the logic formalism may then undergo additional transformations during the course of constructing a proof.

Third, we require that the system have a control structure that determines which transformations to apply, when a solution has been obtained, or when further effort is futile.

Finally, we require that all the above be accomplished with a reasonable degree of computational efficiency.

In the remainder of this chapter, we will consider a number of distinct formalisms for reasoning, and describe how these formalisms are applied to problem solving. Some of the issues addressed include:

- What is reasoning, and what is its role in intelligent behavior?
- How can a reasoning system use a formal language to represent things and their relationships in the world, and how can it solve problems using such a representation?
- What are the conceptual and practical limits of problem solving systems employing formal representations?
- How can a reasoning system deal with imprecisely formulated problems?
- How can a reasoning system select the best representation for a given problem?
- How can a reasoning system know which facts in its database are relevant to solving a given problem?
- How can a machine formulate a plan of action to achieve a desired goal?

HUMAN REASONING

Until the twentieth century, logic and the psychology of thought were considered to be one and the same. In Chapter 1 we quoted Boole's statement as to the purpose of his book on logic: *to investigate the fundamental operations of the mind by which reasoning is performed.* Thus, it is not surprising that often formal logic or probability theory is taken as the ideal, and human reasoning is found to deviate from this ideal. This point of view is in contrast to investigations in visual perception and language, where the biological system is taken as the exemplar and an attempt is made to attain similar performance by machine.

As discussed in Chapter 2, almost nothing is known about the physical machinery used by the brain to carry out its reasoning activity. Attempts to gain insight into the functional aspects, if not the actual brain mechanisms involved in human reasoning have motivated a large body of psychological research. However, unlike experiments in which the speed or accuracy of a perceptual or motor action can be objectively measured, experiments in reasoning are subject to contextual conditions and variables that are difficult to control, and that can only be quantified using subjective judgment. For example, since subjects come to such experiments with a lifelong experience of cooperativeness in conversation, they expect to encounter a cooperative experimenter who will provide them with information useful for solving the posed problem. Thus, although the experimenter may have provided redundant or misleading information, subjects will attempt to use this material to find a solution. There is also the problem of experiments that are foreign to the natural reasoning processes used by people, resulting in misleading conclusions. Scribner [Scribner 77] describes some of the fascinating cultural influences on logical processes.[2] Finally, the experiments often require that human subjects describe their reasoning activities as they solve a problem; the recorded protocols are then analyzed. Such protocol analysis suffers from the fact that people typically do not have access to the reasoning mechanisms that they are *really* using.

Much of the research on human judgment and reasoning is based on the study of "errors." This approach is similar to the study of optical illusions to understand the principles of visual perception or the study of forgetting to learn about memory. Research on systematic errors and inferential biases in reasoning can sometimes reveal the psychological processes that govern judgment and inference. Such research can also indicate which principles of logic and statistics are nonintuitive or counterintuitive. However, given the large body of work investigating human problem solving, there have been surprisingly few results concrete enough to be suitable for transfer to machine-based formalisms.

Human Logical Reasoning

Some of the rules of formal logic are quite intuitive for people, but many others are not. Experiments [Rips 77] have shown that people readily use forms of inference such as "From (P implies Q) and P you can deduce Q." For example, "If John is good, he will be rewarded. John is good. Therefore, John will be rewarded." However, the valid deduction "From (P implies Q) and (not Q) you can deduce (not P)" is mistrusted by people untrained in formal logic. For example, "If John is good, he will be rewarded. John will not be rewarded. Therefore, John is not good." The difficulty with this form may be due to the fact that people are not used to reasoning about what is not true. In addition, people tend not to seek negative information when carrying out reasoning processes.

People have difficulty with many deduction forms, "syllogisms," that deal with "all" and "some." For example, the invalid syllogism "Some A's are B's; some B's are C's implies that some A's are C's" is considered correct by most people. Figure 4-1 shows that there are situations for which this syllogism is false. The valid syllogism "Some B's are A's; No C's are B's; therefore some A's are not C's" was considered as invalid by 60% of tested subjects [Anderson 80].

[2] For example, suppose a subject is presented with the statements, *All women in Biranga are married. Mary lives in Biranga*, and is asked "Is Mary married?" In some cultures, subjects might reply that they cannot answer because they do not know Mary. Others will not accept the initial premise because they know that there are unmarried women in Biranga.

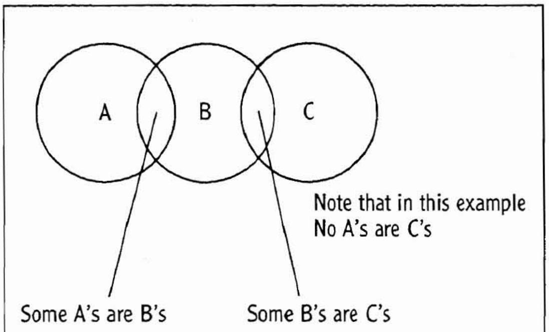


FIGURE 4-1
The Use of a Venn Diagram to Test a Syllogism.

Testing the syllogism "Some A's are B's; some B's are C's; therefore some A's are C's. " The diagram gives an instance in which the syllogism is false.

Johnson-Laird and Wason summarize the situation as follows [Johnson-Laird 77]: "There is not much of a consensus about the psychological mechanisms underlying deduction or even about so fundamental a matter as to whether or not human beings are basically capable of rational inference. . . . [p.76]." In later work, described by Gardner [Gardner 83, pp. 363–367], Johnson-Laird offers the theory that people reason by sequentially integrating the premises and conclusion of an argument into one or more "mental models" which are then searched for inconsistencies (i.e., any interpretation in which the premises lead to a denial of the conclusion). If no such inconsistencies can be found, the conclusion is accepted as valid; while formal logic provides systematic methods for searching for counter-examples, ordinary human reasoning employs no such methods.

Human Probabilistic Reasoning

There are major differences between human and formal probabilistic reasoning [Tversky 74]. When dealing with questions concerning the probability that object A belongs to class B, or the probability that event A originates from process B, a person generally evaluates the degree to which A is representative or resembles B, while ignoring prior probabilities, the effects of sample size, and a statistical principle known as "regression to the mean."[3]

In a typical experiment, subjects were shown brief personality descriptions of several individuals, allegedly sampled at random from a group of 100 persons—engineers and lawyers. For each description, the subjects were asked to assess the probability that it belonged to an engineer rather than to a lawyer. One set of subjects were told that there were 30 engineers and 70 lawyers, and the other group of subjects were told that there were 70 engineers and 30 lawyers. Ignoring the prior probabilities, the two sets of subjects came up with essentially the same probability judgments.

Other experiments have shown that people expect that a sequence of events generated by a random process will represent the essential characteristics of that process even when the sequence is short. For example, in tossing a fair coin, people regard the sequence H-T-H-T-T-H to be more likely than the sequence H-H-H-T-T-T [Tversky 74]. Thus, people expect that the essential characteristics of

[3] This is the phenomenon that exceptional performance is more often than not followed by disappointing performance and failures by improvement.

the process will be represented, not only globally in the entire sequence, but also locally in each of its parts.

Another heuristic used by people is assessing the probability of an event based on the ease with which instances or occurrences can be brought to mind. For example, one may assess the risk of heart attack among middle-aged people by recalling such occurrences among one's acquaintances. Concering this "availability" heuristic, Tversky and Kahneman have pointed out [Tversky 74]:

> Lifelong experience has taught us that, in general, instances of large classes are recalled better and faster than instances of less frequent classes; that likely instances are easier to imagine than unlikely ones; and that the associative connections between events are strengthened when the events frequently co-occur. As a result, man has at his disposal a procedure, the availability heuristic, for estimating the size of a class, the likelihood of an event, or the frequency of co-occurrences, by the ease with which the relevant mental operations of retrieval, construction, or association can be performed. However, this valuable estimation procedure results in systematic errors.

In both the logical and statistical domains, it appears that the human reasoning process is context-dependent, so that different operations or inferential rules are required in different contexts [Hayes 77]. Consequently, human reasoning cannot be adequately described in terms of context-independent formal rules. Furthermore, performance is dramatically improved when an experimental task is related more clearly to the subject's experiences. The difficulty in solving a posed problem is often not intrinsic to the logical structure of the task, but rather to the mode of presentation (e.g., [Johnson-Laird 77]).

FORMAL REASONING AND PROBLEM SOLVING

Requirements for a Problem Solver

A problem exists when there are conditions that certain objects must satisfy, given some set of constraints or facts. A solution is a way of satisfying the conditions. Thus, given the problem of getting from home to the airport, the database of facts should contain information concerning the transportation available, the time available, the distance to be traversed, etc. The condition to be satisfied is your presence at the airport, and the solution, the "plan," is the sequence of operations that you use in satisfying the condition consistent with the given (or implied) constraints.

An obvious first step in solving a problem is to recognize that a problem exists. However, prompt recognition that a problem exists may not be simple. For example, novice chess players may not notice that they are being drawn into a losing position until it is too late. A child may not realize that the bicycle is going too fast until it goes out of control. One might drive for a considerable distance before recognizing that one is lost.

Once the problem is recognized, the problem solver must represent it in a suitable formalism and then plan a course of action using this representation and a knowledge of the effects of proposed actions. As indicated in Chapter 3, the heart of the problem-solving process consists of choosing the right representation, and being able to set up the appropriate correspondence between the problem and the representation.

In carrying out a plan, the problem solver must know when the information at hand is inadequate and should be supplemented or supplanted. If the environment changes, the problem solver must change the plan accordingly. For example, in the airport problem, we may plan to take Bayshore Highway but find that construction has slowed traffic too much, requiring us to take First Street instead.

Any proposed solution must satisfy the conditions of the problem. However, in real-world problems we find that some of the most important conditions are not stated. In the airport problem, the important condition—"I want to get to the airport in a reasonable amount of time"—may not be stated. The solution "walk there" may satisfy the stated conditions but not the implicit ones.

Categories of Reasoning

There are many different systems of reasoning that can be used to solve problems. We divide these systems into the three major categories presented below; however, we will encounter some reasoning techniques that span more than one category.

Deductive Reasoning. In deductive reasoning, we attempt to find a "deductive chain" of "valid" assertions leading from statements which are assumed to be true, to some given assertion whose validity we wish to establish. The power of the deductive approach lies in the fact that the rules of deductive inference obtain new true statements from existing ones. Deductive arguments are characterized by their logical necessity; the conclusion is "entailed" by (implicit in) the premises.

Deduction is meaningful only in the context of a formal system in which symbols are combined and transformed under a given fixed set of rules. The essence of a deductive system is the maintenance of validity or consistency: a statement and its contradiction cannot both be derived. We are therefore guaranteed validity of derived results. However, to insure this property, deductive systems are often extremely awkward in expressing certain types of information. Thus, deductive logic systems have no practical way of dealing directly with probabilistic assertions, or with information implied by quantitative assertions requiring numerical computation; mathematical systems have no practical way of dealing directly with conflicting or probabilistic assertions, or with qualitative statements (e.g., "Bill looks quite a bit like John"); probabilistic systems, to the extent that they can be considered to be deductive, have no practical way of expressing relational information (e.g., "Bill is twice as tall as John"), and no effective way of manipulating assertions that are strictly either true or false.

Inductive Reasoning. In inductive reasoning (nondemonstrative inference), a form of reasoning basic to scientific inquiry, we attempt to find some generalization or abstraction that describes or categorizes a set of data. A major distinction between deduction and induction is that in induction we have a set of constraints to satisfy, rather than an explicit (given) assertion to establish. Further, inductive problems are less likely to be as precisely formulated as deductive ones.

For example, given the problem of finding the next number in the sequence $<1,2,4,8,16>$, most people will give the answer "32" without requiring any additional problem specification. It is typical of problems in induction that there is more than one acceptable answer to the problem (any answer could be justified in the above example). Such problems often require extrapolation, and generally do not permit a definitive way to check the "correctness" of a final answer. Premises support, but do not logically entail the derived solution. The rules of inductive inference do not provide an assured means for deriving new true statements from existing ones.

One of the most important distinctions between deductive and inductive reasoning is the amount of "evidence" that must be invoked to derive a new assertion (or verify some hypothesis). Because of assured consistency and computational considerations, deductive systems generally use long reasoning chains consisting of small steps; in each step, only a very small subset of the total set of "facts" known to the system is explicitly invoked. Deductive systems make "local" syntactic transformations—they cannot take a "global" perspective in solving a problem. On the other hand, because of the possibility of erroneous information, inductive systems use short reasoning chains consisting of big steps. Inductive systems generally attempt to explicitly use as much of their available information as possible in every step since they depend on consensus to insure "correct" conclusions. Thus, inductive systems must work at a global level in solving a problem.

Analogical Reasoning. In analogical reasoning, we set up a correspondence between the elements and operations of two distinct systems. Typically, one of the systems is well understood, and the other is the one we wish to ask questions about; we answer the questions by posing them in the system we understand. An example of analogical reasoning is the solution to the 15 game by the known procedure for playing tick-tack-toe, as described in Chapter 3. Another example is using our knowledge and intuitions about fluid flow to reason about the flow of electrical current.

The major problem in reasoning by analogy is to find the correspondence between the known and unknown systems. For example, if we have an analogy, "An electric battery is like a reservoir," it is not the size, shape, color, or substance of a battery that is relevant, but rather that both store potential energy and release energy to provide power. Thus, only relationships dealing with the storage and release of energy would be meaningful in this analogy. The insight used by a person to recognize that a previously encountered situation is analogous to another situation eludes mechanization.

Common-sense reasoning, discussed later in this chapter, combines analogical and inductive techniques to solve everyday problems about the behavior of physical objects in the world. Analogical reasoning also plays an important role in learning, as will be shown in Chapter 5.

The following sections describe a number of different reasoning formalisms. No matter which formalism is employed, a major part of the reasoning process is the conversion of some given problem into that formalism. This conversion or translation step is actually a problem in analogical reasoning for which we have no adequate solution at present; i.e., we still consider the translation step to be a creative process.

THE DEDUCTIVE LOGIC FORMALISM

In this section we will discuss a special kind of reasoning called "logical deduction," in which true conclusions result when "rules of inference" are applied to true statements. Thus, we are interested in consistent systems in which one proposition may be inferred or deduced from other propositions. A deductive system with a consistent set of premises will be consistent in assigning truth-values to conclusions: such a system cannot prove both that B is true and that B is false. Although the words "true" and "false" are used in the continuing discussion, these words do not necessarily mean true or false in the real world. One should think of "true" and "false" as labels or values (truth-values) that have been assigned to statements, regardless of their relationship to the real world.

Below, we will describe how real-world situations are expressed in the notation of formal logic and how to deduce new facts from a given set of premises. We first describe the propositional calculus that allows us to deal with given propositions (sentences), and compositions of such sentences, which must be either true or false. Then we will treat the predicate calculus that allows us to compose true or false sentences from more primitive elements than complete sentences. These two logic systems have been thoroughly investigated and are well understood, but they correspond to a very small part of the reasoning used by people. However, they form the basis of many AI reasoning programs, and are also part of the machinery underlying "logic programming," as typified by the language PROLOG, discussed in Appendix 4-1.

Propositional Calculus

The calculus of propositions deals with statements or sentences of the type "Water boils at 212 degrees Fahrenheit," "The number 3 is an even number," where the first sentence has an associated truth-value designated by T for true and the second one F for false. Sentences will be denoted by capital letters such as P, Q, R. The following "connectives" are used to combine or modify sentences.

Negation. Negation is indicated by a minus sign, e.g. $-P$, and designates "it is NOT the case that P." If P is true, then $-P$ is false; if P is false then $-P$ is true.

Conjunction. The conjunction of two sentences P, Q is true if both P and Q are true. Conjunction is designated by P&Q, read as P and Q, e.g. (the block is made of wood)&(the block is red).

Disjunction. The disjunction of two sentences P, Q is true if at least one of P, Q is true. Disjunction is designated by PvQ, read P or Q, e.g. (Tom is a man)v(Tom is poor). PvQ allows us to express that at least one of the statements is true without saying which one is true.

Implication. Implication, designated as P→Q, asserts "if P then Q," where P is known as the antecedent and Q the consequent. The sentence is false only if the antecedent is true and the consequent is false; otherwise it is true. Note that, unlike the ordinary use of if-then, e.g., "If taxes rise then the market will drop," no causality is inherent in a logical if-then sentence.

A truth-table is a way of specifying the results of assigning all possible combinations of truth-values to a proposition. For the conjunction operation, the truth table is of the form:

P	Q	P&Q
F	F	F
F	T	F
T	F	F
T	T	T

Thus, P&Q is true only if both P and Q are both true. Two expressions are equivalent if (and only if) their truth-tables are identical. For example, to show that P→Q is equivalent to −PvQ, we develop the following truth-tables:

P	Q	P→Q	−PvQ
F	F	T	T
F	T	T	T
T	F	F	F
T	T	T	T

Since the columns P→Q and −PvQ are identical, P→Q is equivalent to −PvQ. Proof by truth-table comparison is generally not practical because if *n* different propositional variables occur in the premises, then a table with 2^n rows must be filled out. A more efficient approach is to use an inference rule such as:

P	the block is heavy
P→Q	if the block is heavy, then the block is hard to move
Q	the block is hard to move

which can be informally expressed as: "if P is true, and if the statement P→Q is true, then we can infer that Q is true. This deductive rule ("modus ponens"), which can be proved by means of a truth-table, can be used to establish proofs without resorting to the truth-table.

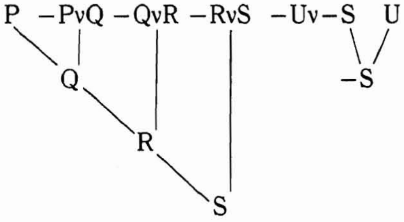
The study of logic involves the study of various inference procedures and the technique of applying these procedures. Until the work of Hao Wang [Wang 60] in 1960, the use of such procedures required intuition, and thus these methods were unsuited for computer implementation. A more recent approach to computer mechanization of logic, called "resolution" [Robinson 65], will be described below.

Propositional Resolution

One can verify by truth-table comparison that the theorem QvS can be proved from the premises PvQ and −PvS. From an operational point of view, we can say that the P and −P terms in the two premises have been eliminated (resolved), leaving a single expression that is the disjunction (logical sum) of the remaining terms. Any proposition can be put in the form (P1)&(P2)&(P3). . ., where P1, P2, and P3 are expressions consisting of disjunctions of variables or negated variables. This transformation can be performed using the propositional equivalences shown in Box 4-1. P1, P2, and P3 are called "clauses," and clauses can be resolved to eliminate variables. For example, the expression (−PvQ)&(P) consists of the clause −PvQ and the clause P. P and −P in these clauses can be resolved to obtain the result Q. Clauses preceded by a negation sign must be transformed to remove the negation sign. A clause such as −(PvQ) must be converted to −P&−Q using De Morgan's theorem given in Box 4-1.

An important approach to theorem proving assumes that the theorem to be proved is false; i.e., its negation is true. Then one shows that this assumption, taken together with the premises, leads to the impossible situation of some variable and its negation both being true (a "contradiction"). Thus, if the negation of the theorem is inconsistent with the premises, the unnegated theorem must be consistent with the premises and therefore true. Arriving at a contradiction is a useful termination condition for an automatic theorem-proving process. An example of a resolution proof is given below.

Given the premise (−PvQ)&(−QvR) &(−RvS)&(−Uv−S), we want to prove the theorem (−Pv−U). Note that this would require a truth-table of 2^6 rows for a truth-table proof. To prove the theorem by contradiction, we take the negation of the theorem, −(−Pv−U), which by De Morgan's theorem in the equivalence table of Box 4-1 is P&U. We place the clauses P, U in the set of clauses (we have placed each in a position that allows the reader to see how the resolution process is carried out):



BOX 4-1 Equivalences in Logic

The following equivalences can be used to convert logic expressions to a standard normal form:

Propositional Calculus

P & Q = Q & P
P v Q = Q v P

− −P = P	Double negation
−(P v Q) = −P & −Q	De Morgan's theorem
−(P & Q) = −P v −Q	De Morgan's theorem

P & (Q v R) = P&Q v P&R

Predicate Calculus

−(*x*)P(*x*) = (E*x*)[−P(*x*)] P does not hold for all *x* = there exists an *x* for which P does not hold.

(*x*)[P(*x*) → (*y*)[Q(*y*)]] = (*x*)(*y*)[P(*x*) → Q(*y*)]

Since we obtain a contradiction, S and $-\mathrm{S}$, for the negation of the proposition, the original proposition $-\mathrm{P} \vee -\mathrm{U}$ must be true, i.e., deducible from the given premises.

Predicates

The propositional calculus is limited in its expressive power; sentences cannot be composed of primitives standing for individual objects and their properties or relationships, but must be composed of primitive elements that are capable of being assigned a truth-value. For example, there is no way of representing an individual such as "John" without making some explicit assertion about him, such as "John is a student." Also, the fact that certain relationships hold for some, or for all individuals, cannot be expressed without being explicit and exhaustive.

In order to provide additional expressive power, the propositional calculus is expanded to the predicate calculus by introducing terms, functions, predicates, and quantifiers, as follows:

Terms or individual variables serve the grammatical function of pronouns and common nouns. They are the things talked about, e.g., "car," "John," or unspecified things such as x, y, or z.

A "predicate" denotes a relationship between objects. A unary relation specifies a property of an object. $\mathrm{Red}(x)$, a unary relation, is a predicate expression that asserts that x is red. Father(John,Tom) asserts that John is the father of Tom. A predicate can take on a value of true or false when its variables have assumed specific values (converting them to terms).

Quantifiers

The universal quantifier, shown by parentheses around the variable,[4] e.g., (x), is the notation that indicates "for all x." Thus, "all men are animals" is expressed as $(x)[\mathrm{Man}(x) \rightarrow \mathrm{Animal}(x)]$. A second quantifier, "there exists," is designated by an E. "There is at least one x such that x is greater than zero" can be represented by $(\mathrm{E}x)(x > 0)$. "A red object is on top of a green one" can be represented by $(\mathrm{E}x)\ (\mathrm{E}y)[\mathrm{Red}(x) \& \mathrm{Green}(y) \& \mathrm{ontop}(x,y)]$.

Universal and existential quantifiers can be combined in the same expression. Thus, "Everyone has a mother" can be expressed as $(x)(\mathrm{E}y)[(\mathrm{Human}(x) \rightarrow \mathrm{Mother}(x,y)]$.

Note that $(\mathrm{E}x)\mathrm{Q}(x)$ allows us to express the fact that something has a certain property without saying which thing has that property, and $(x)[\mathrm{P}(x) \rightarrow \mathrm{Q}(x)]$ expresses the fact that everything in a certain class has a certain property without saying what everything in that class is.

Semantics

Even though we may use symbols that form English words, it must be kept in mind that to an automatic theorem proving system these are merely symbols that are to be manipulated. The system sees no difference between $\mathrm{P}(x)$ and $\mathrm{Red}(x)$; the meaning or semantics must be provided by the user mapping the variables and functions to things in the problem domain. The specification of a domain and the associations between logical sym-

[4] The confusion between parentheses denoting the universal quantifier and those used to denote the variables in a function is easily resolved by context.

bols and the problem domain constitute an *interpretation* or a *model* of the logical system.

Computational Issues

Mechanized inference techniques in the predicate calculus first convert the expressions into a normal form, consisting of propositional-type expressions; the various connectives and quantifiers are removed using the steps shown in Box 4-2. (In logic programming languages such as PROLOG [see Appendix 4-1], the expressions are written directly in a "clause " form, eliminating the need for this conversion.)

In the early 1930s, Herbrand

BOX 4-2 Converting Predicate Calculus Expressions to Clause Form

The following sequence of operations is used to convert a predicate calculus expression to clause form:

1. **Removing implications.** Occurrences of $\mathrm{P}\rightarrow\mathrm{Q}$ are replaced by $-\mathrm{P}\mathrm{v}\mathrm{Q}$. Thus, $(x)[\mathrm{Man}(x)\rightarrow\mathrm{Human}(x)]$ is replaced by $(x)[-\mathrm{Man}(x)\mathrm{v}\mathrm{Human}(x)]$.
2. **Moving negation inwards.** We replace $-[\mathrm{Human}(\mathrm{Caesar})\&\mathrm{Living}(\mathrm{Caesar})]$ by $-\mathrm{Human}(\mathrm{Caesar})\mathrm{v}-\mathrm{Living}(\mathrm{Caesar})$. The quantifier "all" preceded by a negation is transformed as in the example $-(y)[\mathrm{Person}(y)]$ to $(\mathrm{E}y)[-\mathrm{Person}(y)]$. That is, if not all things satisfy a predicate, then there must be at least one thing that does not satisfy it.
3. **Removing the existential quantifiers.** The removal of existential quantifiers, known as "skolemizing," is done by introducing new constant symbols. Instead of saying that there exists an object with a certain set of properties, one creates a name for one such object and simply says that it has the properties. Thus, for $(\mathrm{E}x)[\mathrm{Female}(x)\&\mathrm{Motherof}(x,\mathrm{Eve})]$, we say $\mathrm{Female}(\mathrm{G1})\&\mathrm{Motherof}(\mathrm{G1},\mathrm{Eve})$. When there are universal quantifiers in a formula, skolemization is not quite so simple. If we skolemized $(x)[\mathrm{Human}(x)\rightarrow(\mathrm{E}y)(\mathrm{Motherof}(x,y))]$, "every human has a mother" to $(x)[\mathrm{Human}(x)\rightarrow\mathrm{Motherof}(x,\mathrm{G1})]$, we would be saying "every human has the same mother." Thus, we have to use a function, such as G2 in the expression, $(x)[\mathrm{Human}(x)\rightarrow\mathrm{Motherof}(x,\mathrm{G2}(x))]$, to indicate the dependence of the y on the particular x selected.
4. **Moving universal quantifiers outward.** We can move universal quantifiers outward without affecting meaning. Thus, $(x)[\mathrm{Man}(x)\rightarrow(y)[\mathrm{Woman}(y)\rightarrow\mathrm{Likes}(x,y)]]$ can be transformed to $(x)(y)[\mathrm{Man}(x)\rightarrow(\mathrm{Woman}(y)\rightarrow\mathrm{Likes}(x,y))]$.
5. **Conjunctive normal form.** The expression is now transformed so that conjunctions no longer appear inside disjunctions, i.e., we obtain the form (P)&(Q) . . . , where P, Q . . . do not contain &. This normal form is used in propositional resolution.
6. **Clause form.** The formula we now have is made up of a collection of &'s relating things which are either literals or composed of literals connected by v's. If we have something like (A&B)&(C&(D&E)), where A,B,C,D.E represent (possibly complex) propositions that have no &'s in them, then we can ignore the parentheses and write A&B&C&D&E, and we can consider this a collection of clauses A,B,C, . . .

Proof procedures such as resolution can now be invoked in a manner similar to that described for the propositional calculus; the main distinction is due to the possible existence of variables in the predicate clauses which then requires the use of unification to achieve the necessary matching.

[Herbrand 30] proved that if a set of clauses containing variables is contradictory, then there will exist a finite set of variable-free instances of these clauses that can be shown to be contradictory by propositional methods. An efficient procedure for finding such a contradiction was developed in 1965 by J. A. Robinson [Robinson 65]. This procedure makes inferences by the use of "unification" and propositional resolution. Thus, once we have the expression in clause form, we can carry out these procedures to obtain a proof.

Propositional resolution requires that two clauses to be resolved have a common element (literal), negated in one clause and unnegated in the other. Sometimes a constant, another variable, or a function (not containing the variable) must be substituted for some given variable in order to satisfy the above condition. The process of finding substitutions that make two clauses resolvable is called *unification*. An important feature of the resolution method is that it does not require that the clauses being resolved contain only constants, but allows the most general possible form of the variables to be retained consistent with the resolution condition. For example, we can resolve the two clauses $P(c,x)vF$ and $-P(c,y)vG$ by making the substitution $x=y$; we need not assign a specific value to x or y.

Resolution proof procedures are hopelessly inefficient if they have no mechanisms to specify which of the many possible sequences of resolutions to select. Many different techniques have been developed to deal with this problem. For example, the "set of support" strategy takes the first clause to be resolved from the negation of the statement to be proved (because such a step will eventually be required to complete the proof). It further dictates that at least one resolvent in every resolution must be descended from the negation of the statement to be proved, because only such resolutions are relevant. The "linear format" strategy attempts to keep the sequence of resolutions relevant by requiring that each new resolution make use of the results of the previous one.

A good discussion of the "art" of setting up the proof strategy is discussed in Wos [Wos 84]:

> The use of an automatic reasoning program is an art, even though the program employs unambiguous and exacting notation for representing information, precise inference rules for drawing conclusions, and carefully delineated strategies to control those inference rules. . . . In using an automated reasoning program, one makes good choices for the representation, for inference rules, and for strategies. . . . Without strategy, an automated reasoning program will drown in new information. With strategy, a reasoning program can sometimes perform as a brilliant assistant or colleague.

Nonstandard Logics

In the first order predicate calculus it is not possible to represent relationships among predicates, temporal relationships, hypothetical assertions, beliefs, assertions of possibility, and vague asssertions based on incomplete information. In addition, there is no mechanism for deleting statements from the database. There is a growing literature devoted to the creation and exploration of alternative logics and associated inference mechanisms. Some of these systems are extensions designed to supplement standard logic, while others are alternatives to standard logic. These systems, being explored for use in AI, are described briefly below; a detailed treatment is given in Turner [Turner 85].

Modal logic is concerned with concepts of necessity and possibility. It extends standard logic by using the operators "it is necessary that" and "it is possible that." This type of logic can be used to deal with the concept of "belief," an important consideration in the planning of actions. A modal logic suitable for representing knowledge and action has been developed by Moore [Moore 85].

Temporal logic deals with the representation of time, important in automatic planning and in diagnosis. Concepts such as *is true, was true, will be true,* and *has always been true* must be expressed, as well as time-interval relationships such as *during, before,* and *overlaps in time.*

Higher order logic can represent properties of predicates or even properties of properties of predicates. For example, in the second order predicate calculus, equality can be defined as

$$(P)(x)(y)\ [x=y] \rightarrow [P(x) \rightarrow P(y)],$$

i.e., if x and y are equal, then for all predicates, the predicate of x equals the predicate of y. This quantification over predicates is not permissible in the first order predicate calculus.

In higher order logic, care must be taken to avoid contradictions of the sort discovered by Russell and treated by the theory of types in Russell and Whitehead's *Principia Mathematica*. Higher order logic has not as yet seen much use in AI.

Multivalued logics. While classical logic employs two truth-values, a multivalued logic can represent intermediate values. Multivalued logics are useful for situations in which one cannot always make a commitment to either true or false, and yet one wants a deductive system that is consistent.

Fuzzy logic. In fuzzy logic, predicates such as "red" and "tall" are considered as vague predicates, and an element is considered to have a "grade of membership" in any given set. Truth-values are also considered to lie on a scale between true and false. "A is small; A and B are approximately equal; therefore, B is more or less small"is an example of a fuzzy inference.

Nonmonotonic logic. In classical logic, the system increases its stock of truths as knowledge is added and as inferences are made. There is no mechanism for discarding information or revising beliefs. This aspect of classical logic is termed "monotonic." In nonmonotonic systems, inferences can be made on the basis of available data, but these inferences can be rejected and new ones made when new data become available.

INDUCTIVE REASONING

In inductive reasoning we form generalizations that characterize a class of data from the characteristics of a set of samples of the class. These generalizations, and the inferences based on them, are inductive because it is always possible that our initial conclusions will be invalidated by new evidence, acquired by observing a larger sample, or even a single new sample. Despite this risk, induction is an indispensable mode of reasoning, used continually in everyday life as well as in the development of scientific theory.

In this section we describe the Bayesian and Shafer-Dempster probabilistic formalisms; these formalisms are important tools used in inductive reasoning. While the deductive systems described previously cannot deal with conflicts in evidence because such conflicts lead to logical contradictions, probabilistic techniques are able to make predictions in the presence of conflicting evidence. These predictions will not always be true, but they are good guesses that make effective use of the given information.

Just as there are various forms of deductive reasoning, various forms of probabilistic reasoning are possible. The different forms depend on the nature of the belief measures used and how they are manipulated. Philosophers have identified at least four distinct versions of the concept of probability:

1. The measured frequency of occurrence of events.
2. The disposition of events (or a single event) to occur, e.g., "Everyone who looks at this car agrees that there is a low probability that it will be able to make the trip from New York to Los Angeles"
3. The subjective belief a person has about the likelihood of occurrences of different events
4. The logical relationship between evidence and relevant hypotheses, e.g., "If the patient has a fever and his glypus test is positive, then it is probable that he has Hendrix syndrome"

Probabilistic reasoning first requires the construction of a problem representation. This step, called "sample space construction," or developing the "frame of discernment," formulates the vocabulary and statements that will be used to describe the given problem. Next, a belief "value" is provided for each statement, either by ranking the statements, assigning a belief number to each, or assigning a lower and upper belief number (bound) to each. Finally, the known belief values are combined or pooled, and propagated to modify the belief numbers of other statements, and especially that of the target hypothesis. As in most AI problems, the representation step is crucial. It depends on the designer's understanding of the relevant events in the world, and the availability of evidence that relates to these events. Representation as an issue in probabilistic reasoning is discussed later.

Measures of Belief

There are various characteristics that a belief measure might have. A set of intuitively satisfying characteristics was proposed by Cox [Cox 46] and discussed in detail in Horvitz and Heckerman [Horvitz 86]:

1. **Clarity.** The propositions must be defined precisely enough so that one can tell when a proposition is true or false.

2. **Completeness.** It must be possible to assign a degree of belief to any proposition.
3. **Scalar continuity.** Measures of degree of belief should vary continuously between certain truth and certain falsehood.
4. **Context dependency.** The degree of belief in a particular proposition should depend on knowledge about the truth of other propositions.
5. **Consistency.** If two propositions are logically equivalent, the degree of belief in one proposition given certain evidence should equal the degree of belief in the other.
6. **Hypothetical conditioning.** The belief in the proposition A&B should be a function of the belief in A and the belief in B given that A is true.
7. **Complementarity.** The belief in the negation of A should be determined by the belief in A itself.

Cox showed that these seven properties are logically equivalent to the axioms of classical probability theory; alternative belief formalisms change one or more of these properties. Below we first describe belief revision in classical probability theory, and then discuss the Shafer-Dempster (S/D) theory that rejects several of the above properties.

Bayesian Reasoning

Bayesian reasoning is the classical mechanism used to revise belief, given new evidence [Feller 50, Parzen 60]. We begin with a probability distribution that completely describes our degrees of belief in a set of hypotheses before obtaining new evidence. If a probability P is assigned to an event A, then $(1-P)$ is assigned to $-A$, the nonoccurrence of A. New evidence results in modifying or "conditioning" P based on computations relating evidence to the hypotheses.

In Bayesian reasoning, the logical form of the implication "if E then H," is replaced by "if E then H with a probability P." This "conditional probability" assertion is written $P(H|E)$, and is read "the probability of hypothesis H given that the evidence E is true." Probabilities are updated according to Bayes's theorem:

$$P(H|E) = P(E|H)\ P(H)/P(E).$$

This equation, derived in Box 4-3, states that we can update the probability of hypothesis H, P(H), given that new evidence E, assumed to be true, has been received. Bayesian calculus for complicated situations requires knowledge of the *a priori* probability of some events, e.g. P(H) and P(E), and depends on the sequential use of known conditional probabilities, $P(E|H)$, to evaluate the corresponding values for implied propositions. If the required *a priori* and conditional probability values are known, an evaluation path can be found to allow the computation of the likelihood of some target event. However, the determination of all these necessary *a priori* and conditional values is often impossible or impractical, and one is then forced to heuristic or approximation techniques to compute the unknown values.

When the relationship between events is unknown, a basic technique used is to assume their independence. We can then compute, rather than guess, the *a priori* values of joint events, such as

BOX 4-3 Conditional Probability and the Bayes Theorem

A conditional probability P(H|E) is the probability of an event or hypothesis, H, given that we know that some other event or hypothesis, E, is true. The relationships are readily derived using simple sets. Suppose we have N things, some of them with property H, some of them with property E, and some with both properties E and H as shown in the Venn diagram in Fig. 4-2.

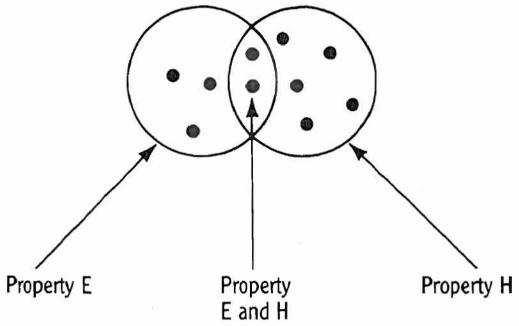


FIGURE 4-2
Example of Conditional Probability Calculation Using the Venn Diagram Representation.

P(H|E)=N(E and H)/N(E)
N(E)=5
N(E and H)=2
P(H|E)=2/5

The conditional probability P(H|E) = N(E and H)/N(E), where N(E and H) is the number of elements that have both properties E and H, and N(E) is the number of elements with property E. If we divide the top and bottom of the right-hand side by N, the total number of elements, we get

$$P(H|E) = \frac{\frac{N(E \text{ and } H)}{N}}{\frac{N(E)}{N}} = \frac{P(E \text{ and } H)}{P(E)}. \qquad (4.1)$$

Using a similar argument, we can obtain the expression for P(E|H), the probability of the event E, given that event H is true:

$$P(E|H) = P(H \text{ and } E)/P(H). \qquad (4.2)$$

Since P(H and E) is the same as P(E and H), we can solve for P(H and E) in Eq. (4.2), P(H and E) = P(E|H)P(H), and substitute in Eq. (4.1) to obtain Bayes's theorem,

$$P(H|E) = P(E|H)P(H)/P(E). \qquad (4.3)$$

This says that if we have an initial (*a priori*) probability of event or hypothesis H, P(H), and we know that event E is true, then we can get an updated probability, P(H|E), assuming that we also know both the probability of event E, P(E), and the conditional probability of E given that H is true, P(E|H).

P(A,B). Thus if A and B are independent P(B|A)=P(B), so that P(A,B)=P(A)P(B|A) =P(A)P(B). Obviously, we get incorrect answers if we assume independence when it is not appropriate.

A second technique is to employ the *principle of insufficient reason*. This principle states that if there is no reason to believe x to be more or less likely than y, then assume that the probability of x equals that of y. Thus, in the absence of any additional information, the probability assigned to any one hypothesis becomes a function of the number of hypothetical alternatives. This is an undesirable property since a particular hypothesis can always be broken down into several subhypotheses, thus altering the *a priori* probabilities assigned to other hypotheses. For example, given that an apple has disappeared in a locked room containing two men and a woman, we might assign to each of two competing hypotheses a probability of 1/2:

P1: A man ate the apple.
P2: A woman ate the apple.

However, we also could have formulated our hypotheses as:

Q1: Bill ate the apple.
Q2: Bob ate the apple.
Q3: Mary ate the apple.

Under the principle of insufficient reason, we assign Q1, Q2, and Q3 each a probability of 1/3. P2 and Q3 are identical assertions which are assigned different probabilities under the principle of insufficient reason simply because of the way we chose to express the remaining alternatives. Even if we could improve the value assignment process for the priors, we might still want to profess ignorance. However, Bayesian probability offers no mechanism to permit this option.

Belief Functions

The Shafer-Dempster belief function formalism [Shafer 76, Garvey 81] rejects the completeness assumption that asserts that a degree of belief can be assigned to any proposition. This approach also rejects the principle of insuffient reason or any probability assignment that provides a value to a proposition when not enough is known about the proposition. In addition, separate measures are assigned to a proposition and its negation, rejecting the complementarity assumption.

In the Shafer-Dempster formalism, the evidence received by a knowledge source results in its apportioning a "unit of belief" among a set of propositions. The amount of belief (called "mass") committed to a proposition represents a judgment as to the strength of the evidence that specifically favors that proposition. It is not required that belief not committed to a given proposition should be given to the negation of the proposition. This is in contrast to the Bayesian approach in which a unit of probability must be apportioned between the two sides of every question.

The Shafer-Dempster formalism makes lack of information concerning probabilities explicit by expressing the belief in a proposition as a subinterval [support(a), plausibility(a)] in the unit interval [0,1]. Using this notation, a proposition would be written A[.25,.85] to indicate that the probability of A is between .25 and .85. The lower value represents the support for a proposition A, and sets a minimum value for its likelihood. The upper value denotes the plausibility of A and establishes a maximum likelihood. The support may be interpreted as the total positive effect a body of evidence has on a proposition, while plausibility represents the total extent to which a body of evidence fails to refute a proposition. The degree of uncertainty about the probability value for a proposition corresponds to the width of its interval. Table 4-1 presents some additional examples of belief assignment employing the Shafer-Dempster representation.

To use belief functions, one partitions evidence into relatively simple components, makes probability judgments separately with respect to each of these components, and then combines these judgments to obtain a final judgment that represents the total evidence.

TABLE 4-1 ■ Examples of Belief Assignment in the Shafer-Dempster Approach

[0,1]	no knowledge about the proposition
[0,0]	proposition is false
[1,1]	proposition is true
[.25,1]	evidence provides partial support for proposition
[0,.85]	evidence provides partial refutation of proposition
[.25,.85]	evidence is conflicting, providing both evidence for and against the proposition

The Shafer-Dempster formalism has the desirable property that the intervals become points when precise probability information is available, and under certain independence assumptions, the corresponding computations produce results consistent with Bayesian probability theory. If the only events that can occur are known to be either true or false, then the results of the Shafer-Dempster computations are consistent with those of deductive logic. The Shafer-Dempster formalism also provides a way of dealing with conflicting information, but now the resulting likelihoods can no longer be interpreted in the same way as Bayesian probabilities, and in particular, they no longer have a simple frequency of occurrence interpretation.

The underlying representation for the Shafer-Dempster formalism consists of an exhaustive list (L) of mutually exclusive event possibilities with each subset (A of L) interpreted as the "proposition" that the true state of the world is one of the elements of L contained within A. L is called a "frame of discernment," to emphasize that each possibility in L can always be split into more specific possibilities, i.e., the resolution with which we view the world can be increased, thus increasing the number of propositions that L discerns.

Single Belief Functions in the S/D Formalism. A "knowledge source" distributes a unit of belief across a set of propositions for which it has direct evidence in proportion to the weight of that evidence as it bears on each proposition. For example, if there are five possible events, a knowledge source KS1 might distribute belief as <.13, .22, .35, 0, 0> for the events A, −A, B, C, D. Once mass has been assigned to a set of propositions, the evidential intervals can be determined directly. Support for a proposition A is the total mass ascribed to A or to its subsets; the plausibility of A is one minus the sum of the mass assigned to −A or to the subsets of −A; the ignorance about A is equal to the mass remaining. For example, the evidential interval for the event A given by KS1 is [.13, .78], since the mass of −A is .22.

Intuitively, mass is attributed to the most precise propositions a body of evidence supports. If a portion of mass is attributed to a proposition, it represents a minimal commitment to that proposition and all of the propositions it implies.

Composition of Individual Beliefs. There is a formal rule for combining belief functions, known as "Dempster's rule." Dempster's rule combines a belief function (constructed on the basis of one item of evidence) with a second belief function (constructed on the basis of another, *assumed independent* item of evidence), so as to obtain a belief function representing the combined body of evidence. An example of this procedure is given in Box 4-4. Dempster's rule makes explicit the fact that the S/D formalism has not escaped a critical weakness of all inductive methods, i.e., the need (for

BOX 4-4 Combining Two Sets of Evidence Using Dempster's Rule

Dempster's rule for combining two different sets of evidence can best be understood by example. Suppose there has been a hit and run accident, and there are two witnesses. The witnesses are willing to assign belief as to the culprit as follows:

Witness #1: I'd say it was a Ford with belief .3
I'd say it was a Chevy with belief .5
I don't know how to distribute .2 of my belief.

Witness #2: I'd say it was a Chevy or a Toyota with belief .7
I don't know how to distribute the remaining .3 of my belief.

We illustrate Dempster's rule using the following tableau:

Witness #1:

Undistributed .2 belief	Chevy or Toyota 0.14	Undistributed 0.06
Ford .3	null 0.21	Ford 0.09
Chevy .5	Chevy 0.35	Chevy 0.15
	Chevy or Toyota 0.7	Undistributed 0.3

Witness #2

The belief distribution for witness #1 is shown on the left of the array, and for witness #2 at the bottom of the array. We multiply the row and column weights to obtain the values assigned to each element of the array. For example, the value of the upper left array element is 0.2 * 0.7 = .14 . Whenever the evidence is incompatible, e.g., the belief of witness #1 that it was a Ford is incompatible with the belief of witness #2 that it was a Chevy or Toyota, we assign that product to the null set. When there is undistributed belief of one witness, we assign that cell the value of the other witness, e.g., undistributed belief for witness #1 causes the upper left cell to be assigned the Chevy or Toyota classification of witness #2.

When we add up all the areas, we obtain:

Pooled Ford values = 0.09
Pooled Chevy or Toyota = .14
Pooled Chevy = .35 + .15 = .50
Pooled uncertain = .06
Conflict (null set) = .21

Since there is 0.21 unit of conflicting mass, we normalize the mutually consistent pooled values so that they sum to 1.0 by dividing each by (1 - 0.21), to obtain:

Ford pooled belief = .11
Chevy or Toyota pooled belief = 0.18
Chevy pooled belief = 0.63
Uncertainty = 0.08

The evidential interval for Chevy is [.63, .89], indicating an uncertainty interval of .26.

These final results agree with our intuition. We would expect that the belief about the Ford would be decreased since witness #2 mentioned nothing about a Ford. The belief for a Chevy has been increased somewhat above the .5 belief of witness #1, since both witnesses have some degree of belief in the Chevy.

either practical or theoretical reasons) to *assume* independence of observations and/or events.

Representing a Problem in a Probabilistic Formalism

As in the case of logical reasoning, choosing an appropriate representation for a given problem is the creative step in probabilistic reasoning. We will illustrate the nature of the representation problem by an example, first describing the problem formulation and then showing the computations. Our example is inspired by the work of Shafer and Tversky [Shafer 85].

Suppose we have an election prediction service and wish to estimate the probability of candidate Jones being elected. There are many facets of election politics that might be considered, including the amount of campaigning effort, the effect of the world situation, the condition of the economy, etc. Suppose our election experts decide that "campaigning effort" is the most crucial determinant. They might set up the alternative strategies for a candidate: (1) maintain current level of campaigning, (2) slightly increase current level of campaigning, (3) greatly increase level of campaigning, and (4) decrease current level. Notice that the experts must supply an estimate of the probability of each event and of the effect that the event will have on winning the election.

Suppose it was felt that the other leading contender, Smith, should be taken into account in making the estimate. We might believe that candidate Smith's campaigning effort could be described by using three levels of activity. Now the interaction of Jones's and Smith's campaign must somehow be computed. Estimates must be made of the probability of activity for Smith and the conditional probabilities of Jones winning, given various levels of activity for both Jones and Smith. At this point an important (and perhaps unrealistic) assumption must be made if we are to avoid estimating how Jones and Smith will respond to each other's strategies. We must assume that the campaign level of each candidate is independent of the activity of the other. A Bayesian computation for this situation is shown in Box 4-5, and an evidential reasoning approach is shown in Box 4-6.

After examining the results of the analysis, the analysts might make other partitions of the problem, using more or fewer levels of campaign activity, or introducing other campaign factors. For each of these it would be necessary to supply the required probability or belief measures.

Comments Concerning the Probabilistic Formalism

Our nominal view of the world embodies the concept that there is a unique course of events that can be characterized by observed or measured physical quantities. Our understanding of the world is, in turn, characterized by our ability to predict the values of these "observables." Physical theories (models, paradigms) can be ranked in terms of how accurately they perform the prediction task. Reality provides an exact and explicit basis for evaluation of proposed theories.

On the above grounds, we might consider probabilistic models as descriptions of processes and events that we

BOX 4-5 An Example of Bayesian Analysis

A four-level breakdown of campaign activity for Jones and a three-level breakdown of campaign activity for his opponent, candidate Smith, is shown in the table. Experts have assigned probabilities to each of Smith's activity levels. Entries in the table specify conditional probabilities that Jones will win, given the activity of Jones and the activity of Smith. Thus, a conditional probability of 0.3 in the upper left element of the table indicates that if Jones maintains his campaign and Smith increases his campaign, then there is a 0.3 chance of Jones winning. A conditional probability of 1 indicates a sure win for Jones, while a 0 indicates a sure loss. The table indicates that a candidate who decreases his campaign activity is going to be in trouble unless his opponent also does so.

	Probability of activity level				
	.85	Maintain	.3	.5	1.0
Jones	.03	Increase slightly	.5	.6	1.0
	.07	Increase much	.7	1.0	1.0
	.05	Decrease	0	0	.5
	Probability of activity level →		.1	.7	.2
	Political activity level →		Increase	Same	Decrease
				Smith	

The probability of Jones winning is the sum of the probabilities of his winning for each of the twelve situations described by the above table. The computation is known as a total evidence design since the final probability is the sum of the probabilities of all the possible situations. The probability for each pair of activity levels is determined by using the formula,

Prob(Jones wins|condition A and condition B)*Prob(A and B),
where P(A and B) = Prob(A|B)*Prob(B).
The independence assumption allows us to say that P(A|B) = P(A).
Thus, Prob(Jones wins) = (.3)(.85)(.1) + (.5)(.85)(.7) + . . . + (.5)(.05)(.2).
Prob(Jones winning) = .586

resort to when deterministic models are not available; they are necessarily cruder, but should be capable of being ranked in terms of accuracy on the same scale as the deterministic models.

Given two probabilistic descriptions of the same situation, such as provided by the Bayesian and Shafer-Dempster formalisms, we might expect to be able to compare their relative performance and choose one or the other as being more accurate. Thus, in the case of the election examples, presented in Box 4-5 and Box 4-6, the Bayesian formulation tells us that candidate Jones has a .586 probability of winning the election, while the Shafer-Dempster formulation tells us that the likelihood of candidate Jones winning is between .239 and .96 (both predictions were based on the same evidence). All this seems quite straightforward, except that the numbers produced by the two formalisms do not really mean the same thing, nor can they be directly compared or evaluated. Suppose, for example, that Jones wins the election. This fact cannot be used to favor either the Bayesian or the S/D approach since neither estimate has a clear meaning that is decisively verified by the real-world result.

Then what do the probabilities really mean? The Bayesian formalism assumes an underlying random process such that, if the election were held often enough under the same conditions, Jones would win 58.6 percent of the time. The Shafer-Dempster formalism provides a way of combining evidence which satisfies our intuition in regard to the ordering of possible outcomes (even when the evidence sources conflict), but does not always have a simple interpretation in terms of the relative frequencies of the outcomes of a random process. Thus, the Bayesian and Shafer-Dempster formalisms provide different underlying models of reality—they are not directly comparable, nor is there generally any way to choose

BOX 4-6 An Example of Belief Function Analysis

Referring to the Bayesian analysis of the two-candidate problem (Box 4-5), we choose to interpret the *a priori* probabilities for different levels of campaign activity as degrees of belief. Thus the degrees of belief for the four hypotheses concerning candidate Jones are (.85, .03, .07, .05). For candidate Smith we have (.1,.7, .2) as the degrees of belief for the three hypotheses concerned with his campaign activity. From Box 4-5 we see that Jones will win when the table entry has the value 1.0. We convert these 1 entries to the proposition "Jones will win" denoted by "Win" in the table on the right. The 0 entries are replaced by the proposition "Jones will lose" denoted by "Lose," and everything else by the proposition "We can't assign a win or lose judgment" denoted by "?"

We now combine these beliefs by Dempster's rule, again assuming independence. The use of Dempster's rule is similar to that shown for the Ford/Chevy/Toyota example of Box 4-4. We add up the products of compatible beliefs. For example, in the upper right-hand element, we have beliefs for Jones and Smith that both agree on the event "Win." We therefore get a contribution of .85*.2 toward that event. Adding up the areas pertaining to a win for Jones we get .85(.2) + .03(.2) + .07[(.7) + (.2)] = .239 .

Adding up the areas that support the proposition "Jones loses" we get (.05)(.1 + .7) = .04 .

	Degrees of belief	Political activity level			
	.85	Maintain	?	?	Win
Jones	.03	Slight increase	?	?	Win
	.07	Large increase	?	Win	Win
	.05	Decrease	Lose	Lose	?
	Degrees of belief →		.1	.7	.2
	Political activity →		Increase	Same	Decrease
				Smith	

This results in an evidential interval for Jones of [.239, .96], indicating a small support, a small refutation of the event "Jones will win," and a very large degree of uncertainty remaining. Thus, we are unable to choose a likely winner in this election.

These conclusions are weaker than the conclusion of the Bayesian analysis, since we are not claiming to have evidence about what will happen in the cases where our descriptions of Smith's and Jones's behavior do not determine the outcome of the election.

between them in unconstrained real-world situations.

ADDITIONAL FORMALISMS FOR REASONING

There are some forms of reasoning that involve combinations of the deductive/inductive/analogical paradigms. Below, we describe some of these: mathematics, programming systems, "production systems," and common-sense reasoning.

Algebraic/Mathematical Systems

In the algebraic/mathematical approach to reasoning, we start with a set of mathematically described (physical) relationships relevant to some (real-world) situation; the problem information is then phrased in terms of these known relations to provide a set of equations; the equations are solved using the standard techniques of mathematics.

Thus, solving a problem such as "If one person can do a job in 3 hours and another can do the same job in 5 hours, how long will it take for them to do the job together?" requires the following steps:

- We must know that the appropriate basic relationship is "rate of doing work times the time worked equals the amount of work done."
- We must assume that working together does not change the individual rates of work.
- We must reason that if a person can do a job in N hours, he does 1/N of the job in 1 hour. Thus, in the given problem, the first person works at a rate of 1/3 of the job per hour, and the second at a rate of 1/5 of the job per hour.
- Time is represented by the variable t. We finally can write the equation $(1/3)t + (1/5)t = 1$
- The equation can now be solved for t using algebra.

Note that the difficult aspect of this type of reasoning consists of (a) knowing that the pertinent relationship is "rate times time equals work done," (b) that people are assumed by convention to work at a constant rate in this problem context, and then (c) translating the problem statement into these algebraic relations. These steps are quite difficult to automate in a general problem-solving context. However, if we know beforehand the types of problems that will be encountered, if the problem language is simple enough, and if no superfluous information has been provided, then we can write a program that solves such word problems by looking for "key words" (see Box 4-7).

Heuristic Search

One form of reasoning is to search through all possible alternatives for a solution to a problem. We often use this approach in our daily lives. For example, we misplace an object and search from location to location in an attempt to find it. Note that we do not blindly explore everywhere, rather we only search in the most probable locations for it. Problems are often amenable to solution by search, provided that there is some organized way of ruling out alternatives that have little probability of being a successful solution. Many AI techniques are based on heuris-

BOX 4-7 Solving Algebraic Word Problems by Computer

The STUDENT program, developed by Bobrow in the late 1960s [Bobrow 68], solves algebraic word problems phrased in natural language. STUDENT sweeps through the input statements several times, carrying out a different transformation on each pass until suitable algebraic equations are obtained. The equations are then solved.

The words and phrases of the problem are considered to be in one of three classes:

Variables. Words that name objects. One important problem that has to be dealt with is how to determine when two different strings refer to the same variable (e.g., at one point the problem might state "...John's money" while at another point the problem might ask "...how much is Tom's money and how much is John's.")

Substituters. These are words and phrases that are replaced to obtain a more standard representation, e.g., "twice" is replaced by "2 times."

Operators. These are words or linguistic forms that represent functions. One simple operator is "plus" which indicates that the two variables surrounding it are to be added.

An appreciation for the procedures used can best be gained from a printout of the various passes made by the program on a typical problem:

```
The original problem to be solved is:
(THE SUM OF LOIS' SHARE OF SOME MONEY
AND BOB'S SHARE IS $4.50.
LOIS' SHARE IS TWICE BOB'S. FIND BOB'S AND
LOIS' SHARE.)
After substitutions the problem becomes:
(SUM LOIS' SHARE OF SOME MONEY AND BOB'S
SHARE IS 4.50 DOLLARS. LOIS' SHARE IS 2 TIMES
BOB'S. FIND BOB'S AND LOIS' SHARE.)

After words have been tagged by function, the
problem is:
((SUM/OP) LOIS' SHARE (OF/OP) SOME MONEY
AND BOB'S SHARE IS 4.5 DOLLARS (PERIOD/
DELIMITER) LOIS' SHARE IS 2 (TIMES/OP. 1) BOB'S
(PERIOD/DELIMITER) (FIND/QUESTIONWORD)
BOB'S AND LOIS' SHARE (PERIOD/DELIMITER)

Converted to simple sentences:
((SUM/OP) LOIS' SHARE (OF/OP) SOME MONEY
AND BOB'S SHARE IS 4.5 DOLLARS (PERIOD/
DELIMITER)
(LOIS' SHARE IS 2 (TIMES/OP 1) BOB'S (PERIOD/
DELIMITER)
((FIND/QUESTION WORD) BOB'S AND LOIS'
SHARE (PERIOD/DELIMITER)

Converted to equation form:
(EQUAL (LOIS' SHARE) (TIMES 2 (BOB'S)))
(EQUAL (PLUS (LOIS' SHARE OF SOME MONEY)
(BOB'S SHARE)) 4.5 DOLLARS)

However, these equations were insufficient to find a
solution. The program then assumes:
((BOB'S) IS EQUAL TO (BOB'S SHARE))
((LOIS' SHARE) IS EQUAL TO (LOIS' SHARE OF
SOME MONEY))

A solution can then be obtained:
(BOB'S IS 1.5 DOLLARS)
(LOIS' SHARE IS 3 DOLLARS)
```

Note that since the system could only make a partial match on the name of the variables, it assumed that a partial match, e.g., BOB's to BOB's SHARE, was equivalent to a complete match. This allowed a solution to be obtained.

Thus, STUDENT is a system for dealing with a restricted class of problems, but it is very effective in this limited domain.

tic search procedures, rule-of-thumb techniques that direct the search process to the more attractive candidates for solution. Procedures that search for valid proof sequences, discussed in various parts of this chapter, are typically controlled by heuristic rules.[5]

Programming Systems that Facilitate Reasoning and Problem Solving

Conventional programming languages require the user to specify.procedures that are to be carried out on the data. The flow of control, and the tests to be performed must be explicitly described. However, programming systems have also been designed to accept nonprocedural "programs," i.e., there are systems that permit the user to state his goal or intent, and the built-in mechanisms of the system attempt to devise procedures to attain these goals. Such systems are often written in programming languages that facilitate writing programs whose purpose is to reason and solve problems.

A formal algorithm for carrying out a reasoning procedure could be implemented in any one of the many programming languages that provide symbol storage, matching, combining of strings or lists, and some type of conditional branching operation. AI problem solving programs are more concerned with manipulating strings of symbols, e.g., rearranging symbols or substituting one symbol for another, than with numerical computation, e.g., multiplying two numbers together. Special languages designed for AI programming have therefore been developed—the most popular being LISP and its dialects. A brief description of LISP is given in Appendix 4-1. In addition, many AI problems have the characteristic that after a certain amount of progress is made toward a solution, a dead end is reached, and the program must "backtrack," returning certain variables to their original state. This requires much bookkeeping activity that is extraneous to the "logic flow" of the solution for the given problem. The logic-based language PROLOG, described in Appendix 4-1, provides deductive procedures and automatic backtracking.

In a typical program, even one written in LISP or PROLOG, the flow of control and the utilization of data are specified by the program's code, but in "pattern-directed inference systems" (PDIS), the processing modules are activated by patterns in the input data or in the "working storage." A module is inactive until a certain data pattern or situation exists, at which point a response is made. The module's activity typically consists of adding or deleting data in the working store. Such a system is "data driven" rather than "program driven," and "programming" in such a system consists of specifying the pattern to be matched by each module and the corresponding action to be taken.

The system is controlled by software that handles the tasks of pattern matching, monitoring database changes, and carrying out the actions specified by the active modules. Typically, the control structure of the system is given, and the investigator supplies the specifications of the modules.

An important type of PDIS is the

[5] The subject of heuristic search techniques is discussed extensively in Nilsson [Nilsson 71] and in Pearl [Pearl 84].

"rule-based" or "production" system, discussed further in Chapter 7, in which each module is a rule that has a left-hand side containing the pattern templates that must be satisfied, and a right-hand side that specifies the actions to be carried out. Because the rules are kept separate from the control structures, it is possible to modify rules without requiring any programming changes to the rest of the system. The OPS-5 production language presented in Appendix 4-1 is an example of a rule-based system. A typical rule is of the form [(A AND B)→C], which specifies that if both A and B appear in the input or working storage, then C will be entered into the working storage. Entering a new fact or assertion by satisfaction of the left-hand side of a rule is called "forward chaining" or "antecedent driven" reasoning. It is also possible to interpret the same rule as "if we want to establish C, then it is first necessary to establish both A and B." This is known as "backward chaining" or "consequent driven" reasoning. Backward chaining is often used to set up a goal tree that directs the search for needed data items.

Practical production systems consist of many rules, typically several hundred to a few thousand, and have been applied to a variety of applications, most notably in the form of "expert systems" (see Chapter 7). Systems such as OPS-5 depart from the "pure" PDIS by providing features that permit the programmer to exercise a considerable degree of control over the processing.

Common-Sense Reasoning

The reasoning techniques that we have dealt with in this chapter use representations of numerical quantities and propositions, i.e., formalisms based on the concept of number and on the algebra of sets. However, we have not yet discussed another type of reasoning used by people; their impressive ability to reason using common-sense theories of the world—their everyday beliefs about what the world is like. Such reasoning appears to be qualitative in nature. For example, consider the reasoning used in answering the following question: "What happens if we turn on the water tap in the bathtub, with the plug in the tub?" We reason as follows. For some time the level of the water will rise, until it reaches the top of the tub. The water then flows over the sides of the tub, and covers the bathroom floor. After the bathroom floor is covered to some level, the water will flow to other rooms and will leak into the floor, dropping onto any room below. If some of the water finally escapes from the house, and it is cold enough outside, the water may freeze, possibly into icicle-shaped forms.

Devising a qualitative theory of liquid behavior, and developing an associated reasoning formalism is extremely difficult, since one must first deal with a coherent body of water, then, as it overflows, some of the water separates from the main body, forming a new body of water on the floor, followed by the conversion to individual drops as it falls into the room below. Somehow the formalism has to deal with the creation of new objects from old, the qualitative physics of water flow, and the interaction of water with gravity forces, physical surfaces, temperature of the environment, etc.

Some of the issues that arise in trying to represent and reason about common-sense knowledge are as follows:

Representing common-sense knowledge. In order for an intelligent entity to deal with everyday things, it must have a database consisting of descriptions of these things. The database would have to include descriptions of general spatial properties, the behavior of materials and liquids, and have a "naive" understanding of topics such as physics, botany, zoology, ecology, etc. For example, the database would have to capture the properties of water, including properties when it is still, slowly moving, or energetically moving. The behavior in each of these activity states depends on whether the water is flowing on a surface, contained, or unsupported. In addition, the formulation must consider whether the water is in bulk form or divided (as in a mist), and the time-history of the situation.

A collection of papers describing efforts to formalize common-sense knowledge is contained in Hobbs [Hobbs 85].

Qualitative reasoning. A special type of reasoning seems to be involved in dealing with everyday objects. Although the real world is continuous to our senses, a person does not have to possess continuous representations, such as those typically provided by mathematics and physics, to deal with this world. It seems that people deal with the world by treating it qualitatively using only a few values for any of the variables, e.g., very big, big, medium-sized, small, very small. Similar quantizations may be employed for nearness, strength of forces, weights, etc. Reasoning based on this type of vague quantization seems to be adequate for solving everyday problems, for being able to tell how something works, or using something in a way for which it was not intended, e.g., using a fallen tree as a seat. Formalisms for qualitative physics and common-sense reasoning about causality are described in De Kleer [De Kleer 84] and Kuipers [Kuipers 84].

Relevance. Given a real-world situation, how can a reasoning system determine which other objects will have a significant interaction with the current object of interest? We are (again) faced with the relevance problem in trying to determine what aspects of what objects, in the whole universe, should enter the reasoning process.[6]

PROBLEM SOLVING AND THEOREM PROVING

Previous sections described a variety of reasoning techniques; this section will discuss how these techniques can be used to solve problems. Basically, the approach is to:

(a) Represent the concepts, relationships, and constraints of the task environment in the formalism required by the problem solver.

(b) Apply the solution techniques mechanically by operating on the representations; the "meaning" of the

[6] The problem of relevance is a vital part of the gestalt psychologist view of problem solving as originally formulated by Max Wertheimer [Wertheimer 61].

expressions is neither required nor used by the problem solver.

The power of any general problem solving approach is that a large number of interesting problems can be cast into some common form. However, converting the problem to this form is often the main step in obtaining a solution. Once the problem is in the required form, the role of the computer can generally be viewed as equivalent to searching a decision (or game) tree to find a required node or best path.

At the present time there are many classes of problems that (for practical reasons) cannot be put into the form required by existing machine-based general problem solvers. Some examples are: scene analysis problems, in which the machine must describe or understand a real-world scene; language understanding problems; and problems for which all the relevant conditions cannot be specified, e.g., artistic creation.

Representing the Problem

To illustrate the nature of the representation issue for the various general problem solving approaches, we will use a classic example, the monkey/bananas problem (the M/B problem):

"A monkey and a box are in a room, and some bananas are hanging from the ceiling, just out of reach of the monkey. What should he do to get the bananas?"

Given just this statement of the problem, a person readily identifies the pertinent operators concerned with moving the monkey, pushing the box, standing on the box, and finally, reaching for the bananas. A person ignores other possible operations such as the monkey throwing the box, kicking the wall, scratching himself, etc. Thus, when we present a mechanical problem solver with only the "relevant" operations, we are greatly simplifying the problem solving effort required. How, then, might the problem be presented to a general problem solving program? The initial conditions are clear:

> The bananas are at location L. The monkey is at location X. The box is at location Y.

The basic operations available could be indicated as follows without giving away the solution:

> The monkey can walk from location x to location y.
>
> If the monkey and the box are at location x, the monkey can push the box from location x to location y, or he can climb the box.
>
> If the monkey can reach the bananas, he can grab them.

The crucial question that now arises is: how can we specify reachability of the bananas? In a neutral way we might say:

> The bananas are 6 feet off the floor.
>
> The reach of the monkey is 5 feet.
>
> The box is 2 feet high.
>
> If the monkey stands on the box his reach will be extended by the height of the box.

An alternative formulation, and one that gives the problem away is:

> If the monkey stands on the box his reach is within the height of the bananas.

An even more blatant form is:

> If the box is under the bananas and the monkey stands on the box, then he can reach the bananas.

We will show how the problem can be represented for the most blatant form of the problem statement using the predicate calculus, the PROLOG logic programming language, OPS-5 (a production rule system), and the general problem solver (GPS) formalism. The intent is to illustrate the nature of these formalisms in a simple problem situation. Each of the approaches must deal with the frame problem, i.e., the problem of knowing what things in the world change as a result of an action. For example, if the monkey was at location *b* and moves to location *c*, a reasoning system must determine what objects have changed their location (e.g., the monkey's pants, but not necessarily the box he was standing on).

The Predicate Calculus Representation for the Monkey/Bananas (M/B) Problem

The representation for a predicate calculus approach to the monkey/bananas problem is given in Appendix 4-2, as described in Nilsson [Nilsson 71b]. For his exposition, Nilsson simplifies the problem by ignoring the need for the monkey to go to and remain with the box, and we will follow his example.

The frame problem is handled by using the concept of *state,* e.g., the box is considered to be at a certain location, b, in a particular state, s,: AT(box, b,s). "States" and "objects" are represented by state variables and object variables, respectively. Relations between objects, and properties of states and actions are indicated using "situational fluents" which are functions that include states among their arguments, and whose result is also a state. An operation carried out on an object can be viewed as changing it from one state to another. For example, if the monkey climbs the box, we can consider it to be in a new state of "on-boxness." Given a time sequence of operations carried out on an object, we can say that the various operators caused the object to transition from state to state. The proof procedure must find the sequence of operators that will convert the initial state in which the monkey does not have the bananas to the state in which he does. This final state is given in terms of the sequence of states that produced it, thus indicating the sequence of operations that must be used to obtain the end result. A good proof procedure will avoid blind alleys and explore only paths that seem promising.

The initial state is described by -ONBOX(s0), the monkey is not on the box at the initial state s0. The bananas are at location C. The question now posed is "does there exist a state such that the monkey has the bananas?," or formally, (EXISTS s)HAS_BANANAS(s). The predicate calculus solution using resolution, is given [Nilsson 71b] as

> HAS_BANANAS[GRASP(CLIMB_BOX(PUSH_BOX(C,s0)))].

Note the role of the state variable in describing the sequence of operators:

1. Pushing the box to C starting in initial state, s0, causes the new state

PUSH__BOX(C,s0), and we can call the new state s1.

2. The CLIMB__BOX operator then causes a new state, CLIMB__BOX(s1), which we call s2.
3. GRASP(s2) results in a new state s3.
4. Finally, HAS__BANANAS(s3) is the desired solution.

The predicate calculus expression that describes the effect of GRASP provides most of the solution, since the problem solver is specifically told that the monkey should be on the box and the box should be at the location of the bananas in order for the monkey to grasp the bananas. The "solution" is the sequence of operations that will satisfy the needed conditions for GRASP.

PROLOG Representation of the M/B Problem

The PROLOG representation of the monkey/bananas problem is given in Appendix 4-2. The frame problem is handled by retracting old and asserting new database items, e.g., at(monkey,b) is retracted when the monkey moves to c, and at(monkey,c) is asserted. The order of statements in the program is unimportant, except when two rules deal with the same goal (then, the first one encountered will be used). However, the order of terms within statements is crucial, since the analysis of the right-hand side proceeds from left to right. Thus, if we set up the overall goal in the following manner,

```
hasbananas :- at(bananas,X),
move(box,X), move(monkey,X),
onbox (X).
```

we are stating that wherever the bananas happen to be located, that should also be the location of the box. The system will first instantiate the value of X for at(bananas,X). It will then have the ideal goal when it attempts to process the next clause, at(box,X), since it will force the location of the box to be at the same location as the bananas. If we were to reverse the terms,

```
hasbananas:-
move(box,X), at(bananas,X), etc.,
```

the move(box,X) goal will cause a non-productive and semi-infinite search as the system tries all possible values of X.

Notice also, that in the hasbananas top-level goal, the use of the same variable forces the onbox operation to be carried out only under the bananas. This prevents the monkey from getting on the box every time his location was the same as the box. Many such subtle "cheats" are scattered throughout the program.

Production Rule (OPS-5) Representation for the M/B Problem

A production rule representation of the monkey/bananas problem, using OPS-5, is given in Appendix 4-2. The frame problem is handled by the "remove" and the "make" operations. A set of production rules is used for GO, PUSH, CLIMBON, and GRAB, that cause the monkey to move, push the box, climb on the box, and grab the bananas, respectively. Note that the set of rules for PUSH forces the monkey to move the box to where the bananas are. The rule says that if the monkey and the box are at location 1 and the bananas are at location 2, then make location 2 the location of the monkey and the box. The GO and PUSH rules occur before the CLIMBON rule, and therefore set things up so that although CLIMBON is satisfied, these other rules take priority until the monkey and the box are under the bananas. CLIMBON is thus prevented from firing before the appropriate situation is obtained, avoiding the embarrassing outcome of a monkey trapped on the box, but not under the bananas, with no operator to remove him. The careful arrangement of the rules can be thought of as a way of implicitly programming the desired state sequence. Because the behavior of the system can be quite sensitive to the order of the rules, the designer may have to program the system by entering special conditions to keep certain rules from firing at the "wrong" time. For a complete (70 pages) exposition of how the M/B problem can be handled in OPS-5, see Brownston [Brownston 85].

General Problem Solver Representation for the M/B problem

The general problem solver (GPS) [Ernst 69] was a system developed in the 1960s in which problem solving is carried out by reducing the differences between the current state and a goal state, an approach known as "means-ends analysis." To use GPS on a problem, it is necessary to specify the objects and the operators for transforming the objects. An initial state and a goal state are also specified. The specificiations must include how the differences between states are to be measured, and how the procedures to be used relate to state differences. "Programming" in GPS consists of providing these specifications.

The representation for the GPS approach to the monkey/bananas problem is given in Appendix 4-2. This formulation was originally presented by Ernst and Newell [Ernst 69]. The task environment includes the operators to be used (CLIMB, WALK, MOVE_BOX, and GET_BANANAS), the "pretest" conditions for their actuation, and the effects of the operators. The "differences" that must be considered between the present state of the world and what one would like it to be are given, along with the difficulty of reducing each difference. Finally, the specific task is given, including the ultimate goal and the initial state.

Probably the most significant information given is the quantification of the difficulty of reducing each difference. This is the implicit control information that enables the system to solve the problem. Since the difference between the goal state and contents of the monkey's hand is indicated as the most difficult problem, GPS tries to eliminate that difference, and it must create a subgoal to accomplish this. Since the next most difficult difference is associated with the location of the box, it attempts to satisfy this subgoal. Notice that the box being under the bananas is a specific pretest for getting the bananas into the monkey's hand. The box location goal is satisfied by causing the monkey to move the box to the desired location. The monkey's place pretest indicates that the monkey must be on the box in order for the monkey to get the bananas. This then causes the monkey to climb onto the box. Note that without the given difference ordering, the monkey would climb the box whenever he was at the box. If a way of climbing down was provided, then the monkey would cycle at this point.

Formalisms or Reasoning Systems?

In the above examples, we have illustrated that the M/B problem can be solved in each of the major deductive formalisms previously discussed. It was also noted that a valid solution would not be obtained if there were slight alterations in how the problem was presented, or in how the operators were defined and ordered. It is clear that these deductive formalisms are not "reasoning systems" in the full sense of this term, (see the definition of reasoning in the introduction to this chapter), but rather a framework for problem solving in which human understanding and intervention is still a necessary ingredient. The human must "bias" the mapping of the problem into the selected formalism so that the "syntactic" transforms invoked by the formalism operate in a highly constrained search space known to contain the desired answer. The pigeon and the banana problem, an amusing analog to the monkey/bananas problem taken from the field of psychological experimentation, is presented in Box 4-8.

Relating Reasoning Formalisms to the Real World

Formal systems for reasoning are constructed to achieve specific goals such as completeness and consistency. Because of the means used to achieve these goals, there will often be a mismatch between the formal system and the type of expressions and reasoning used by people. For example, a formal system will assign "true" to the implication "If the moon is made of Swiss cheese, then France is a country," since this is of the logical form "false implies true." However, most people expect there to be a relationship between the two parts of the implication, and would consider this example inane. Even the conjunction AND does not

BOX 4-8 The Pigeon and The Banana Problem

We have indicated the various ways in which the designers had to give away the solution to allow their programs to solve the monkey/bananas problem. The following study concerning problem solving by pigeons (*Nature*, March 1, 1984), shows that what we had been calling the monkey/bananas problem was actually the *pigeon and the banana problem.*

The researchers first trained four pigeons to push a box toward a green spot at the base of a cage wall. The birds did not push when the spot was removed. Next, the animals were trained to climb onto a box and peck at a banana placed overhead. Each bird was occasionally placed alone with the banana until the bird neither flew nor jumped toward it. The pigeons were able to solve the feeding problem; they pushed a box placed at the edge of the cage until they could climb onto it and peck at the banana.

Several other pigeons were trained to peck at the banana but were not taught to climb onto the box; to climb and peck but not push the box; and to climb, peck, and push the box, but not toward a target. These birds also learned not to jump or fly toward the banana. But none of them could solve the feeding problem.

The successful birds had to be given all the explicit steps needed to solve the problem; they were only required to put together the correct sequence.

translate directly to the logical form; e.g., in the sentence "John AND Mary are a happy couple," "couple" cannot apply to John or Mary individually. (We cannot conclude that John is happy AND Mary is happy.)

There are many real-world concepts about causality, imagined or fictitious events, verb tenses, imperative forms, and modal forms, to name only a few, that are readily expressed in natural language and are reasoned about by people, but are difficult to capture in any of our existing formalisms.

DISCUSSION

We have described the nature of "problems," and formalisms for reasoning about problems. The difficulty of converting even well-posed problems into a suitable formalism has been indicated; the difficulties of converting ill-posed problems are even more overwhelming. Indeed, one might consider intelligent behavior as the ability to strip away nonessential elements from a problem to allow application of a suitable problem-solving approach.

This chapter has concentrated on the problem-solving machinery once the problem representation process has been carried out. In a way, this is like looking under the lamppost for an object that has been lost at night somewhere else. Unfortunately, we are forced into this stance because most of the AI work in mechanized reasoning has dealt with the formal (proof) machinery, and not with the automatic problem conversion process.

There is still much controversy concerning the role of logic and deductive inference in common-sense reasoning. One view is that logic can be used for analysis of knowledge, but not for reasoning by intelligent agents. The other view claims that logic is the only approach that offers: (1) an assured procedure for deriving new facts from known or assumed truths, (2) the ability to say that an existentially quantified proposition is true without knowing exactly what object makes it true, and (3) the ability to reason by cases.

It was shown that the logic representation can be thought of as providing a language for making assertions about the world; various deductive formalisms can then operate on this representation to answer questions, devise plans, and solve problems. However, the computational feasibility of the deductive process is strongly dependent on the way that the assertions are expressed, and the nature of the external guidance that has been provided. Combinatorial explosion must be avoided, since all of the formalisms have a worst case computational cost that increases exponentially with the number of initial assertions.

Although there are various strategies incorporated into theorem provers to improve the efficiency of the proof-finding process, there are no effective purely syntactic mechanisms that can direct an automatic proof system to select only those statements that are relevant, but still adequate, to obtain the desired proof. If we have a large database, many unproductive paths are typically pursued, and an enormous number of inappropriate deductions carried out.

In a very important sense, deductive systems have to be "programmed" if they are to avoid the necessity for the equivalent of exhaustive search: the user must understand, and supply to the system, some approximation to the solution of the problem to be solved. There is thus an equivalence between what has been called the "automatic programming problem," and automatic problem solving by deductive systems. Since very little progress has been made in finding a general solution to the automatic programming problem, we should not expect currently available deductive systems to be capable of functioning autonomously as general problem solvers.

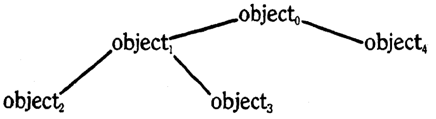
Appendixes

4-1

AI Programming Languages

The LISP Programming Language

In programming computers for artificial intelligence applications, one is often required to represent arbitrary objects and the relationships among them. This is in contrast to other computer applications where numerical computation is the main theme. The LISP language, designed in 1958 by John McCarthy of Stanford University, has become the primary language used in AI. (Some of the present-day variants include INTERLISP, FRANZLISP, MACLISP, COMMONLISP, and ZETALISP.) Simple lists, such as (object1 object2 object3), and more complicated structures, such as



can be uniformly represented. LISP commands permit the programmer to extract elements from lists, to combine lists in various ways, and to carry out mathematical and logical operations. A conditional branching function and facilities for extensive data structure manipulation are also provided.

One characteristic of LISP that is often puzzling to the novice is that procedural knowledge is expressed as a composition of nested functions. Rather than having a program consisting of a series of sequential steps, as in most conventional languages, in LISP the desired operations are expressed in the form of a single complex function that is composed of simpler functions. Also, much use is made of recursion in which the function calls itself. This is illustrated in the following LISP program for factorial.

```
(Define                                   ;define a function
        (factorial (Lambda (N)            ;factorial is the name of the
                                          ;function, and the argument is N
        (cond ((zerop N) 1)               ;if N=0, then result=1 and return
        (T (times N (factorial (sub1 N)   ;otherwise, N*factorial(N-1)
]                                         ;close with required number of
                                          ;right parentheses
```

A more complex problem programmed in LISP is given on the following page.

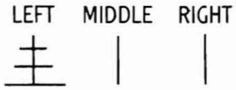
There are many reasons for the success of LISP: it was the first available programming language having the needed flexibility for AI problems, and it became the language of choice in university AI centers. However, a more important reason is that excellent programming environments were developed for the language, consisting of powerful sets of highly integrated editing and debugging tools. An important feature of these environments was that LISP code was interpreted and the programmer could see the results of executing a portion of such code immediately, without having to go through a tedious compilation process. Thus, LISP provided an interactive environment in which all data and functions could be inspected or modified by the programmer. An error in a function or data object could be corrected, and the correction tested, without the need to recompile the program. It is LISP's interactive environment that allows massive programs to be developed one "layer" at a time.

Another useful feature is the dynamic allocation of storage: intermediate results from subsidiary functions are passed on to the calling function, but are not retained after they are used. Thus, the system can automatically recover the memory storage that was used in obtaining the intermediate results, freeing programmers from the responsibility for detailed memory management.

Finally, the LISP language syntax is quite simple: a LISP program is a binary tree. This uniformity of syntax and functions permits a LISP program to examine other LISP programs, and to produce additional LISP programs that can be executed.

A recent contribution to the popularity of LISP is the development of personal work stations based on this language. These "LISP machines" have good graphics, powerful computational capabilities, and can be networked to other machines so that results and programs can be shared.

The Tower of Hanoi Problem in LISP. The Tower of Hanoi problem is a good example of the use of recursion and of the type of thinking that goes into representing a problem in the LISP language. We are given three pegs, LEFT, MIDDLE, and RIGHT and N disks of decreasing size on the LEFT peg.



The problem is to move the disks one at a time from the LEFT peg to the RIGHT peg without putting a larger disk on a smaller disk. The MIDDLE peg can be used as an intermediate storage location when required. The basic approach is to assume that we can get the top N-1 disks to an intermediate peg. We now can place the remaining large disk on the RIGHT peg. The problem is then to move the N-1 disks to the RIGHT peg. This can be accomplished by repeating the original procedure, i.e., using a recursive approach.

The LISP solution uses a function HANOI(N, SOURCE, DESTINATION, OTHER), where N is the number of disks, SOURCE (where a disk is to be taken from), DESTINATION (where the disks removed from SOURCE are to be placed), and OTHER (the current intermediate storage location). SOURCE, DESTINATION, and OTHER take on the values LEFT, RIGHT, and MIDDLE, in any order. For example, HANOI(1, MIDDLE, LEFT, RIGHT) indicates that a disk is to be removed from MIDDLE and placed on LEFT.

Note 1 says that if we can somehow move N-1 disks from SOURCE to some intermediate peg, OTHER, then (Note 2) the remaining disk, the Nth disk, can be moved

The actual LISP program is:

```
        (HANOI
            [LAMBDA (N SOURCE DESTINATION OTHER)
                (COND
                  ((EQP N 1)                        ;if N=1
                    (PRIN1 "MOVE THE DISK ON")      ;message to user to move
                    (PRIN1 SOURCE)                  ;disk from the current
                    (PRIN1 "TO")                    ;value of SOURCE to the
                    (PRIN1 DESTINATION)             ;current value of DESTINATION
                  )
                (T (HANOI (SUB1 N) SOURCE OTHER DESTINATION) ;Note 1
                   (HANOI 1 SOURCE DESTINATION OTHER)        ;Note 2
                   (HANOI (SUB1 N) OTHER DESTINATION SOURCE) ;Note 3
                ))
        ])
```

from SOURCE to DESTINATION. Then (Note 3) we now transfer the N-1 disks from OTHER to DESTINATION using the original SOURCE peg for intermediate storage.

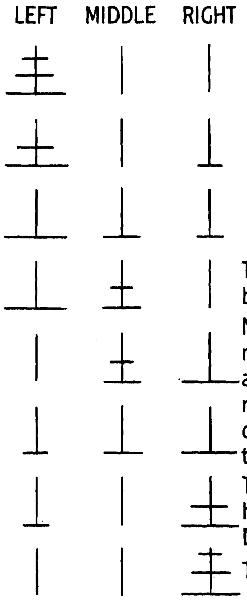
The sequence of operations of the program for the case of 3 disks is shown in Fig. 4-3. The reader is encouraged to work through the LISP program to see how the recursion "unwinds."

The PROLOG Programming Language

Although one can express a problem in predicate calculus and then remove the resulting quantifiers using techniques shown previously, an attractive alternative is to express the logical expressions directly in a quantifier-free clausal form. This is the approach adopted for the programming logic language PROLOG [Clocksin 81]. The motivation for such "logic programming" is that programs will be easier to write (and to read) than programs in a procedural language, since they do not require an explicit statement about how things are to be done, but are more like a specification of what the program should achieve.

The PROLOG clausal form is a restricted subset of the standard form, having the advantage that simple and efficient theorem provers have been developed for it. For some sentences the standard form allows a more economical and natural expression than the PROLOG form. See Kowalski [Kowalski 79] for a comparison.

PROLOG programs consist of (1) declarations of facts about objects and their relationships, (2) rules that define objects and their relationships, and (3) questions about objects and their relationships. A period follows



The top N-1 disks have been moved to the OTHER peg.

Now the largest disk is moved to the DESTINATION peg, and the problem has been reduced to moving the two disks from the CENTER to the DESTINATION peg.

The largest disk on the CENTER has now been brought to the DESTINATION peg.

The solution.

FIGURE 4-3
Solution of the Tower of Hanoi Problem.

every statement in PROLOG. Some examples of PROLOG expressions are given below.

Facts: Some examples of facts in PROLOG are: likes(mary,john).; valuable(gold).; owns(john,gold).; and father(john,mary). Any number of arguments can be used in a fact. To the system, a fact is of the form a(b,c,d, . . .); the mnemonics are merely an aid to the programmer. The programmer must decide what any of the fact expressions mean, e.g., valuable(gold) could mean that a specific piece of gold is valuable, or that in general the mineral gold is valuable.

Questions: Once we have some facts, we can ask questions about them. Thus, the question ?- owns(mary,book). causes PROLOG to look in the database of facts for that fact. If owns(mary,book) is in the database, the system answers "yes," otherwise it responds "no."

Variables: If we want to ask "Who does John like," we express this using a variable, e.g. ?-likes(john,X). If the database contains likes(john,flowers)., PROLOG will respond with X = flowers. The variable X is now "instantiated" to have the value "flowers."

Conjunctions: We can ask "Is there anything that both John and Mary like" by using the expression "?-likes(mary,X),likes(john,X)., where the comma between the facts stands for the conjunction AND. PROLOG first finds an entry of the form likes(mary,something)., and instantiates X to "something." The system then tries to find an entry in the database: likes(john,something). If no such entry is found, then the system backtracks and tries to find another fact that satisfies "likes(mary,X)." If it finds one, then a new value of X is instantiated and the system tries to find "likes(john,X)" for the new value of X. All of this backtracking is performed automatically by the system.

Rules: Rules have the form "(consequence) IF (conditions)," meaning that a certain consequence follows if the conditions hold. For example "likes(john,X) IF likes(X,wine)" indicates that John likes any X, IF X likes wine. Using the PROLOG notation B:- A for B IF A, the relationship "sisterof(X,Y)" can be defined as:

```
sisterof(X,Y) :- female(X),parents(X,m,f),parents(Y,m,f).
```

which says that X is the sister of Y if X is female and if X and Y have the same parents.

Built-in predicates: PROLOG has a set of built-in predicates that provide the programmer a way of expressing control information about how the proof is to be carried out. This is necessary because without such mechanisms PROLOG would spend unacceptable amounts of time trying to carry out proof procedures that are not fruitful. (For example, the "cut" symbol written "!", that allows the programmer to indicate to the system which previous choices it need not consider again when it backtracks through the chain of satisfied goals.)

A simple PROLOG program is given below, along with a target question and a trace of how the program carried out the deduction. Note that each rule can read both in a declarative way and a procedural way. Thus the user can make declarative statements that the system can use in a procedural manner. Both the declarative and procedural interpretations are given only for the first of the two rules.

```
                    % Rule 1
descendant (B,C) :- % declarative: C is a descendant of B
                    % if C is an offspring of B.
   offspring(B,C)   % procedural: To determine that C
                    % is a descendant of B
                    % determine that C is an offspring of B


                    % Rule 2
descendant(B,C) :-  % To determine that C is a descen-
                    % dant of B,
   offspring(D,C),  % determine both that C is an off-
                    % spring of D
   descendant(B,D).% AND that D is a descendant of B

offspring(abraham,ishmael). % This is the database
offspring(abraham,isaac).   % of offspring data
offspring(isaac,esau).
offspring(isaac,jacob).
```

The following trace shows how the question "is esau a descendant of abraham?" is processed. "Call" indicates a rule is to be invoked in an attempt to answer a question or achieve a subgoal. The return is either a "failure," or a successful "exit" with the instantiated variable indicated. The numbered lines indicate the "depth" of the portion of the proof being worked on. For example, line (4) is examining whether isaac is a descendant of abraham, and this then requires that the proof of isaac being an offspring of abraham (5) first be established. The control of the system is goal driven, i.e., the system proceeds from rule to rule as needed to satisfy subgoals. When there is a failure, the system automatically backtracks or tries an additional rule related to the current goal.

```
        | ?- descendant(abraham,esau).
                 We are asking if esau is
                 a descendant of abraham
(1) Call : descendant(abraham,esau) ?
(2) Call : offspring(abraham,esau) ?
(2) Fail  : offspring(abraham,esau)
                 Can't be established by first rule,
                 so now try 2nd rule, 1st part
(3) Call : offspring(_119,esau) ?
                 119 is an i.d. number that PROLOG
                 has used to designate a variable.
(3) Exit  : offspring(isaac,esau)
                 Finds that esau is offspring of isaac
(4) Call : descendant(abraham,isaac) ?
                 Now work on 2nd part of 2nd rule
(5) Call : offspring(abraham,isaac) ?
                 Determine if isaac was offspring of abraham
(5) Exit  : offspring(abraham,isaac)
                 From database, he was
(4) Exit  : descendant(abraham,isaac)
                 From first rule, isaac is a descendant
                 of abraham
(1) Exit  : descendant(abraham,esau)
                 Since both conditions of 2nd rule are
                 satisfied, esau is a descendant of abraham
yes
                 Therefore answer is "yes"
```

OPS-5: A Programming Language for Production Systems

OPS-5 [Forgy 77] is a language for writing production systems programs. If the goal (condition) portion of a rule is satisfied, then the "action" portion causes some change to occur in working memory. One can store items in working memory using the "make" command. "Remove" is used to remove items, and "modify" to change items. Thus the fact that a block named block1 has the color "red" is added to working memory by

```
(make block
     name block1
     color red)
```

A typical production rule would be written:

```
(p find-colored-block         ;p denotes production
     (goal                    ;if there is a goal which
          status active       ;is active to find
          object block        ;a block
          color  <z>)         ;of a certain color
      (block                  ;and there is a block
          color  <z>          ;of that color
          name <block>)       ;with a certain name

      ———————>

      (make result pointer<block>)
             ;then enter a pointer in
             ;working storage that
             ;indicates the name of the
             ;block that satisfies the goal.
      (modify status satisfied)
             ;and change the goal marking
             ;to satisfied
)
```

This says that

> IF in the working memory there is an item known as a goal, and if that goal is to find a block of a certain color, and if there is also an item in working memory describing a block of that color.
>
> THEN make an item called a "result" that points to the block and change the goal item to indicate that it is now satisfied. (The pointer result can then be used by any other rules requiring a block of that color.)

An OPS-5 program consists of a set of such production rules and stored items. The system is activated when new items appear in working memory that cause certain rules to be activated. The activated rules add, delete, and modify items in the memory to cause further activity. Production rule programming requires a different way of thinking than conventional procedural programming.

4-2

The Monkey/Bananas Problem

The Predicate Calculus Formulation

The following predicates and operators are given as part of the predicate calculus formulation of the M/B problem [Nilsson 71a]:

Predicates
ONBOX(s), monkey is on the box in state s
AT(box,b,s), box is at location b in state s,
HAS__BANANAS(s), monkey has bananas in state s
Operators
(It is important to remember that each operator returns a new state value)
GRASP(s), the state attained when grasping bananas in state s,
CLIMB__BOX(s), the state attained when the monkey climbs box in state s,
PUSH__BOX(x,s), the state attained when the monkey pushes box to location x starting in state s.

The preconditions and effects of operators are expressed in the predicate calculus notation:

(It is assumed that the monkey and the box are never separated.)

PUSH__BOX(x,s): If the monkey isn't on the box, in the state s, then the box and the monkey will be at location x in the new state attained by applying PUSH__BOX to state s.

(ALL x ALL s)[-ONBOX(s) → AT(box, x, PUSH__ BOX(x,s))]

CLIMB__BOX: The monkey will be on the box in the state attained by applying the operator CLIMB__BOX to the state s. Note that the argument of ONBOX is a new state, CLIMB__BOX(s).

(ALL s)[ONBOX(CLIMB__BOX(s))]

GRASP: If the monkey is on the box and the box is at C (the location of the bananas) in state s, then the monkey will get the bananas in the state attained by applying GRASP to state s.

(ALL s)[(ONBOX(s) AND AT(box,c,s) → HAS__BANANAS(GRASP(s))]

In addition, it must be stated explicitly that the position of the box does not change when the monkey climbs on the box.

(ALL x ALL s)[AT(box,x,s) → AT(box,x,CLIMB__BOX(s))]

As described in the text, the predicate calculus solution using the above formulation is:

HAS__BANANAS[GRASP(CLIMB__BOX(PUSH__ BOX(C,S0)))].

This solution, and its conversion to a plan that could be used by the monkey to obtain the bananas is described in Nilsson [Nilsson 71a].

APPENDIX 4-2

The Monkey/Bananas Problem in PROLOG

The PROLOG formulation for the monkey/bananas problem is shown below; see text for additional comments. The initial conditions are shown first, followed by the rules.

```
offbox.                     %these are the given initial conditions.
at(bananas,c).              %lower case characters are constants.
at(monkey,a).               %thus, a,b,c are constants that represent
at(box,b).                  %fixed locations of the monkey, the box, and the bananas.

hasbananas :-               %this is the top level goal. It states
    at(bananas,B),          %that the monkey has the bananas when
    move(box,B),            %the box and the monkey have been moved
    move(monkey,B),         %to the same location as the bananas,
    onbox(B).               %and the monkey is on the box.

move(monkey,B) :-           %to achieve the goal of moving the monkey
  ( at(monkey,B);           %to B, either the monkey is already at B, or
    at(monkey,C),           %the monkey is at C and we should establish
       goto(C,B) ).         %goto(C,B) that moves him from C to B.

move(box,B) :-              %to achieve the goal of the box at B
  ( at(box,B);              %either the box is already at B or
    at(box,C),              %the box is at C, and we should establish
       pushbox(C,B) ).      %pushbox to move the box from C to B.

goto(B,C) :-                %to get the monkey from B to C, either
  ( at(monkey(C));          %he is already at C, or he is off the box,
    offbox,
    retract(at(monkey,B)),  %and we then retract his former location
    assert(at(monkey,C)) ). %and assert his new one.

pushbox(B,C) :-             %to push the box from B to C,
    offbox,                 %the monkey must be off the box,
    at(box,B),              %the box must be at B
    move(monkey,B),         %the monkey must be at B,
    retract(at(monkey,B)),  %and we then retract the previous
    retract(at(box,B)),     %locations of the box and the monkey,
    assert(at(monkey,C)),   %and assert the new ones.
    assert(at(box,C)).

climbbox(B) :-              %to establish climbbox,
    offbox,                 %establish that the monkey is off the box,
    move(box,B),            %move the box to B,
    at(monkey,B),           %establish that the monkey is at B
    retract(offbox).        % retract offbox.
```

```
onbox(B) :-               %this merely says that if we establish
    climbbox(B).          %climbbox, we establish onbox. This
                          %statement could be eliminated by replacing
                          %onbox by climbbox in all the other statements.
```

A trace of the operations that occur when we ask to establish hasbananas is given below. The numbers shown on the left refer to depth levels of search, and the numbers such as _85 represent the labels of temporary variables used by the system:

```
| ?- hasbananas.                         top goal
  (1) 0 Call : hasbananas ?
  (2) 1 Call : at(bananas,_85) ?
  (2) 1 Exit : at(bananas,c)             at(bananas,c) established
  (3) 1 Call : move(box,c) ?             trying to move the box to c
  (4) 2 Call : at(box,c) ?
  (4) 2 Fail : at(box,c)
  (5) 2 Call : at(box,_102) ?
  (5) 2 Exit : at(box,b)
  (6) 2 Call : pushbox(b,c) ?            pushbox(b,c) needed to
                                         satisfy move(box,b)
  (7) 3 Call : offbox ?                  trying to satisfy pushbox
  (7) 3 Exit : offbox
  (8) 3 Call : at(box,b) ?
  (8) 3 Exit : at(box,b)
  (9) 3 Call : move(monkey,b) ?          has to move the monkey to b
  (10) 4 Call : at(monkey,b) ?
  (10) 4 Fail : at(monkey,b)
  (11) 4 Call : at(monkey,_143) ?
  (11) 4 Exit : at(monkey,a)
  (12) 4 Call : goto(a,b) ?              using goto to move the monkey
  (13) 5 Call : at(monkey(b)) ?
  (13) 5 Fail : at(monkey(b))
  (14) 5 Call : offbox ?
  (14) 5 Exit : offbox                   establishes that offbox is true
  (15) 5 Call : retract(at(monkey,a)) ?
  (15) 5 Exit : retract(at(monkey,a))
  (16) 5 Call : assert(at(monkey,b)) ?
  (16) 5 Exit : assert(at(monkey,b))
  (12) 4 Exit : goto(a,b)
  (9) 3 Exit : move(monkey,b)            monkey moved to box at b
  (17) 3 Call : retract(at(monkey,b)) ?
  (17) 3 Exit : retract(at(monkey,b))
  (18) 3 Call : retract(at(box,b)) ?
  (18) 3 Exit : retract(at(box,b))
  (19) 3 Call : assert(at(monkey,c)) ?
  (19) 3 Exit : assert(at(monkey,c))
  (20) 3 Call : assert(at(box,c)) ?
  (20) 3 Exit : assert(at(box,c))
  (6) 2 Exit : pushbox(b,c)
  (3) 1 Exit : move(box,c)            box moved to c
  (21) 1 Call : move(monkey,c) ?
  (22) 2 Call : at(monkey,c) ?
  (22) 2 Exit : at(monkey,c)
  (21) 1 Exit : move(monkey,c)
  (23) 1 Call : onbox(c) ?            establishing onbox
  (24) 2 Call : climbbox(c) ?         establishing climbbox
  (25) 3 Call : offbox ?              verifying that monkey is off box
  (25) 3 Exit : offbox                verified monkey off box
  (26) 3 Call : move(box,c) ?
  (27) 4 Call : at(box,c) ?           verifying that box is at c
  (27) 4 Exit : at(box,c)
  (26) 3 Exit : move(box,c)
  (28) 3 Call : at(monkey,c) ?        verifying that monkey is at c
  (28) 3 Exit : at(monkey,c)
  (29) 3 Call : retract(offbox) ?
  (29) 3 Exit : retract(offbox)
  (24) 2 Exit : climbbox(c)           monkey can climbbox
  (23) 1 Exit : onbox(c)
  (1) 0 Exit : hasbananas             monkey has bananas
yes
```

The Production Rule Formulation

In the OPS-5 production rule formalism, a set of productions is used, each of which specifies the items that can appear in working storage, and the actions that will result when these items actually do appear. If more than one production is satisfied by items in working storage, then the production highest on the list will be activated. Thus, the ordering of the productions is important.

In the production rule approach to the monkey/bananas problem, the initial contents of working storage are:

Initial: goal working, at monkey r, at box b, at banana s, on monkey floor

This says that a goal is being worked on, the monkey is at r, the box is at b, the bananas are at s, and the monkey is on the floor.

The set of productions are:

```
go
((goal working)                         ;if we are still working on a goal,
 (at monkey <loc1>)                     ;goal, and the monkey is in loc1,
 (at box (<loc2> <> <loc1>))            ;and the box is at loc2 not equal to loc1,
 (on monkey floor)                      ;and the monkey is on the floor
———————>
 (remove 2)                             ;remove from working storage the
 (make at monkey <loc2>))               ;fact that monkey is at loc1, and replace monkey location with loc2
```

```
push
((goal working)
 (at monkey <loc1>)                    ;if monkey is at loc1 and the box
 (at box <loc1>)                       ;is at the same location
 (at banana (<loc2> <> <loc1>)) ;and the banana is not at loc1
 (on monkey floor)                     ;and the monkey is on the floor
——————>
 (remove 2)                            ;remove ws entry for monkey location
 (remove 3)                            ;remove ws entry for box location
 (make at monkey <loc2>)               ;enter into ws that monkey is at loc2
 (make at box <loc2>))                 ;and so is the box

climbon
((goal working)
 (at monkey <loc1>)                    ;if the monkey is at loc1,
 (at box <loc1>)                       ;and the box is at loc1
 (on monkey floor)                     ;and the monkey is on the floor
——————>
 (remove 4)                            ;delete fact that monkey is on
 (make on monkey box))                 ;floor, and add to ws the fact that monkey
                                       ;is now on box

grab
((goal working)
 (at banana <loc1>)                    ;if the banana is at loc1
 (at box <loc1>)                       ;and so is the box,
 (at monkey <loc1>)                    ;and so is the monkey,
 (on monkey box)                       ;and the monkey is on the floor
——————>
 (remove 1)                            ;goal has been satisfied
 (remove 2)                            ;banana has been removed
 (make monkey has banana))             ;enter result in ws
```

The sequence of working memory states is:

after go: goal working, at monkey b, at box b, at banana s, on monkey floor

after push: goal working, at monkey s, at box s, at banana s, on monkey floor

after climbon: goal working, at monkey s, at box s, at banana s, on monkey box

after grab: at monkey s, at box s, on monkey box, has monkey banana

APPENDIX 4-2

General Problem Solver Representation

In the GPS approach to the monkey/bananas problem, we are given a task environment that specifies the set of places, the operators, the "differences," the difference ordering, and the task. These are as follows:

I. Task Environment
  A. Miscellaneous: the set of places on the floor = (place1, place2, under the bananas)

  B. Operators
    1. CLIMB
      Pretest: The monkey's place is the same as that of the box
      Result: The monkey's place becomes on the box.
    2. WALK
      Variable: x is in the set of places
      Result: the monkey's place becomes x.
    3. MOVE__BOX
      Variable: x is the set of places
      Pretests: the monkey's place is in the set of places
        the monkey's place is the box's place
      Results: The monkey's place becomes x
        The box's place becomes x

(Note: the difference ordering discussed below keeps the monkey from being on the box at this point. Thus, one can omit the test for the monkey being on the floor to move the box.)

    4. GET__BANANAS
      Pretests: The box's place is under the bananas
        The monkey's place is on the box
      Results: The contents of the monkey's hand is "bananas"

  C. Differences (this indicates the kinds of difference that one can have between what is and what should be. For example, the monkey's place may be different than the desired monkey's place.)

    D1 is the monkey's place
    D2 is the box's place
    D3 is the contents of the monkey's hand

  D. Difference ordering (this indicates the order of difficulty in reducing a difference)

    D3 is more difficult to reduce than is D2 which is more difficult to reduce than D1 (Thus, it is more difficult to take care of the difference of the monkey's hand being empty, than it is to change the difference involved with the monkey's location.)

II. Specific Task
  A. TOP GOAL: Transform the Initial OBJ into the Desired OBJ
    ·(i.e. take the situation described by Initial OBJ and somehow attain the Desired OBJ)

B. Objects
   1. Initial OBJ
      a. The monkey's place is place1
      b. The box's place is place2
      c. The contents of the monkey's hand is "empty"

   2. Desired OBJ
      The contents of the monkey's hand is "bananas"

GPS will first note that there is a difference to be reduced with respect to the contents of the monkey's hand when the Initial OBJ is compared to the Desired OBJ. There is no way of achieving this reduction directly, since the pretests for GET__BANANAS are not satisfied. In trying to satisfy these preconditions, GPS will find differences between what is and what should be, and guided by the difference ordering, GPS will choose the next difference to eliminate. The program tries to eliminate the more difficult differences before trying the simpler ones. The various WALK, MOVE__BOX, and CLIMB operators will have to be exercised before the GET__BANANAS operator can be invoked.