PART 2
Knowledge Representation

4
Formalized Symbolic Logics

Starting with this chapter, we begin a study of some basic tools and methodologies used in AI and the design of knowledge-based systems. The first representation scheme we examine is one of the oldest and most important, First Order Predicate Logic (FOPL). It was developed by logicians as a means for formal reasoning, primarily in the areas of mathematics. Following our study of FOPL, we then investigate five additional representation methods which have become popular over the past twenty years. Such methods were developed by researchers in AI or related fields for use in representing different kinds of knowledge.

After completing these chapters, we should be in a position to best choose which representation methods to use for a given application, to see how automated reasoning can be programmed, and to appreciate how the essential parts of a system fit together.

4.1 INTRODUCTION

The use of symbolic logic to represent knowledge is not new in that it predates the modern computer by a number of decades. Even so, the application of logic as a practical means of representing and manipulating knowledge in a computer was not demonstrated until the early 1960s (Gilmore, 1960). Since that time, numerous

systems have been implemented with varying degrees of success. Today, First Order Predicate Logic (FOPL) or Predicate Calculus as it is sometimes called, has assumed one of the most important roles in AI for the representation of knowledge.

A familiarity with FOPL is important to the student of AI for several reasons. First, logic offers the only formal approach to reasoning that has a sound theoretical foundation. This is especially important in our attempts to mechanize or automate the reasoning process in that inferences should be correct and logically sound. Second, the structure of FOPL is flexible enough to permit the accurate representation of natural language reasonably well. This too is important in AI systems since most knowledge must originate with and be consumed by humans. To be effective, transformations between natural language and any representation scheme must be natural and easy. Finally, FOPL is widely accepted by workers in the AI field as one of the most useful representation methods. It is commonly used in program designs and widely discussed in the literature. To understand many of the AI articles and research papers requires a comprehensive knowledge of FOPL as well as some related logics.

Logic is a formal method for reasoning. Many concepts which can be verbalized can be translated into symbolic representations which closely approximate the meaning of these concepts. These symbolic structures can then be manipulated in programs to deduce various facts, to carry out a form of automated reasoning.

In FOPL, statements from a natural language like English are translated into symbolic structures comprised of predicates, functions, variables, constants, quantifiers, and logical connectives. The symbols form the basic building blocks for the knowledge, and their combination into valid structures is accomplished using the syntax (rules of combination) for FOPL. Once structures have been created to represent basic facts or procedures or other types of knowledge, inference rules may then be applied to compare, combine and transform these "assumed" structures into new "deduced" structures. This is how automated reasoning or inferencing is performed.

As a simple example of the use of logic, the statement "All employees of the AI-Software Company are programmers" might be written in FOPL as

$$(\forall x)\ (\text{AI-SOFTWARE-CO-EMPLOYEE}(x) \rightarrow \text{PROGRAMMER}(x))$$

Here, $\forall x$ is read as "for all x" and $\rightarrow$ is read as "implies" or "then." The predicates AI-SOFTWARE-CO-EMPLOYEE(x), and PROGRAMMER(x) are read as "if x is an AI Software Company employee," and "x is a programmer" respectively. The symbol x is a variable which can assume a person's name.

If it is also known that Jim is an employee of AI Software Company,

$$\text{AI-SOFTWARE-CO-EMPLOYEE}(\text{jim})$$

one can draw the conclusion that Jim is a programmer.

$$\text{PROGRAMMER}(\text{jim})$$

The above suggests how knowledge in the form of English sentences can be translated into FOPL statements. Once translated, such statements can be typed into a knowledge base and subsequently used in a program to perform inferencing.

We begin the chapter with an introduction to Propositional Logic, a special case of FOPL. This will be constructive since many of the concepts which apply to this case apply equally well to FOPL. We then proceed in Section 4.3 with a more detailed study of the use of FOPL as a representation scheme. In Section 4.4 we define the syntax and semantics of FOPL and examine equivalent expressions, inference rules, and different methods for mechanized reasoning. The chapter concludes with an example of automated reasoning using a small knowledge base.

SYNTAX AND SEMANTICS FOR PROPOSITIONAL LOGIC

Valid statements or sentences in PL are determined according to the rules of propositional syntax. This syntax governs the combination of basic building blocks such as propositions and logical connectives. Propositions are elementary atomic sentences. (We shall also use the term *formulas* or *well-formed formulas* in place of sentences.) Propositions may be either true or false but may take on no other value. Some examples of simple propositions are

It is raining.
My car is painted silver.
John and Sue have five children.
Snow is white.
People live on the moon.

Compound propositions are formed from atomic formulas using the logical connectives not and or if . . . then, and if and only if. For example, the following are compound formulas.

It is raining and the wind is blowing.
The moon is made of green cheese or it is not.
If you study hard you will be rewarded.
The sum of 10 and 20 is not 50.

We will use capital letters, sometimes followed by digits, to stand for propositions; T and F are special symbols having the values true and false, respectively. The following symbols will also be used for logical connectives

~ for not or negation
& for and or conjunction

V for or or disjunction
→ for if . . . then or implication
↔ for if and only if or double implication

In addition, left and right parentheses, left and right braces, and the period will be used as delimiters for punctuation. So, for example, to represent the compound sentence "It is raining and the wind is blowing" we could write $(R \ \& \ B)$ where R and B stand for the propositions "It is raining" and "the wind is blowing," respectively. If we write $(R \ V \ B)$ we mean "it is raining or the wind is blowing or both" that is, V indicates inclusive disjunction.

Syntax

The syntax of PL is defined recursively as follows.

- T and F are formulas.
  If P and Q are formulas, the following are formulas:

$$\neg P$$

$$(P \ \& \ Q)$$

$$(P \ V \ Q)$$

$$(P \rightarrow Q)$$

$$(P \leftrightarrow Q)$$

All formulas are generated from a finite number of the above operations.

An example of a compound formula is

$$((P \ \& \ \neg Q \ V \ R) \rightarrow (Q \rightarrow S))$$

When there is no chance for ambiguity, we will omit parentheses for brevity: $(\neg(P \ \& \ \neg Q))$ can be written as $\neg(P \ \& \ \neg Q)$. When omitting parentheses, the precedence given to the connectives from highest to lowest is $\neg$, &, V, →, and ↔. So, for example, to add parentheses correctly to the sentence

$$P \ \& \ \neg Q \ V \ R \rightarrow S \leftrightarrow U \ V \ W$$

we write

$$((((P \ \& \ \neg(Q)) \ V \ R) \rightarrow S) \leftrightarrow (U \ V \ W))$$

Semantics

The semantics or meaning of a sentence is just the value true or false; that is, it is an assignment of a truth value to the sentence. The values true and false should not be confused with the symbols T and F which can appear within a sentence. Note however, that we are not concerned here with philosophical issues related to meaning but only in determining the truthfulness or falsehood of formulas when a particular interpretation is given to its propositions. An *interpretation* for a sentence or group of sentences is an assignment of a truth value to each propositional symbol. As an example, consider the statement $(P \ \& \ \neg Q)$. One interpretation (I_1) assigns true to P and false to Q. A different interpretation (I_2) assigns true to P and true to Q. Clearly, there are four distinct interpretations for this sentence.

Once an interpretation has been given to a statement, its truth value can be determined. This is done by repeated application of semantic rules to larger and larger parts of the statement until a single truth value is determined. The semantic rules are summarized in Table 4.1 where t, and t' denote any true statements, f, and f' denote any false statements, and a is any statement.

TABLE 4.1 SEMANTIC RULES FOR STATEMENTS

Rule number	True statements	False statements
1.	T	F
2.	$\neg f$	$\neg t$
3.	$t \ \& \ t'$	$f \ \& \ a$
4.	$t \vee a$	$a \ \& \ f$
5.	$a \vee t$	$f \vee f'$
6.	$a \rightarrow t$	$t \rightarrow f$
7.	$f \rightarrow a$	$t \leftrightarrow f$
8.	$t \leftrightarrow t'$	$f \leftrightarrow t$
9.	$f \leftrightarrow f'$	

We can now find the meaning of any statement given an interpretation I for the statement. For example, let I assign true to P, false to Q and false to R in the statement

$$((P \ \& \ \neg Q) \rightarrow R) \vee Q$$

Application of rule 2 then gives $\neg Q$ as true, rule 3 gives $(P \ \& \ \neg Q)$ as true, rule 6 gives $(P \ \& \ \neg Q) \rightarrow R$ as false, and rule 5 gives the statement value as false.

Properties of Statements

Satisfiable. A statement is satisfiable if there is some interpretation for which it is true.

Contradiction. A sentence is contradictory (unsatisfiable) if there is no interpretation for which it is true.

Valid. A sentence is valid if it is true for every interpretation. Valid sentences are also called tautologies.

Equivalence. Two sentences are equivalent if they have the same truth value under every interpretation

Logical consequences. A sentence is a logical consequence of another if it is satisfied by all interpretations which satisfy the first. More generally, it is a logical consequence of other statements if and only if for any interpretation in which the statements are true, the resulting statement is also true.

A valid statement is satisfiable, and a contradictory statement is invalid, but the converse is not necessarily true. As examples of the above definitions consider the following statements.

P is satisfiable but not valid since an interpretation that assigns false to P assigns false to the sentence P.

$P \vee \neg P$ is valid since every interpretation results in a value of true for $(P \vee \neg P)$.

$P \& \neg P$ is a contradiction since every interpretation results in a value of false for $(P \& \neg P)$.

P and $\neg(\neg P)$ are equivalent since each has the same truth values under every interpretation.

P is a logical consequence of $(P \& Q)$ since any interpretation for which $(P \& Q)$ is true, P is also true.

The notion of logical consequence provides us with a means to perform valid inferencing in PL. The following are two important theorems which give criteria for a statement to be a logical consequence of a set of statements.

Theorem 4.1. The sentence s is a logical consequence of $s_1, \ldots, s_n$ if and only if $s_1 \& s_2 \& \ldots \& s_n \rightarrow s$ is valid.

Theorem 4.2. The sentence s is a logical consequence of $s_1, \ldots, s_n$ if and only if $s_1 \& s_2 \& \ldots \& s_n \& \neg s$ is inconsistent.

The proof of theorem 4.1 can be seen by first noting that if s is a logical consequence of $s_1, \ldots, s_n$, then for any interpretation I in which $s_1 \& s_2 \& \ldots \& s_n$ is true, s is also true by definition. Hence $s_1 \& s_2 \& \ldots \& s_n \rightarrow s$ is true. On the other hand, if $s_1 \& s_2 \& \ldots \& s_n \rightarrow s$ is valid, then for any interpretation I if $s_1 \& s_2 \& \ldots \& s_n$ is true, s is true also.

The proof of theorem 4.2 follows directly from theorem 4.1 since s is a

TABLE 4.2 SOME EQUIVALENCE LAWS

Idempotency	$P \vee P = P$ $P \& P = P$
Associativity	$(P \vee Q) \vee R = P \vee (Q \vee R)$ $(P \& Q) \& R = P \& (Q \& R)$
Commutativity	$P \vee Q = Q \vee P$ $P \& Q = Q \& P$ $P \leftrightarrow Q = Q \leftrightarrow P$
Distributivity	$P \& (Q \vee R) = (P \& Q) \vee (P \& R)$ $P \vee (Q \& R) = (P \vee Q) \& (P \vee R)$
De Morgan's laws	$\neg(P \vee Q) = \neg P \& \neg Q$ $\neg(P \& Q) = \neg P \vee \neg Q$
Conditional elimination	$P \rightarrow Q = \neg P \vee Q$
Bi-conditional elimination	$P \leftrightarrow Q = (P \rightarrow Q) \& (Q \rightarrow P)$

logical consequence of $s_1, \ldots, s_n$ if and only if $s_1 \& s_2 \& \ldots \& s_n \rightarrow s$ is valid, that is, if and only if $\neg(s_1 \& s_2 \& \ldots \& s_n \rightarrow s)$ is inconsistent. But

$$\begin{aligned}\neg(s_1 \& s_2 \& \ldots \& s_n \rightarrow s) &= \neg(\neg(s_1 \& s_2 \& \ldots \& s_n) \vee s) \\ &= \neg\neg(s_1 \& s_2 \& \ldots \& s_n) \& \neg s) \\ &= s_1 \& s_2 \& \ldots \& s_n \& \neg s\end{aligned}$$

When s is a logical consequence of $s_1, \ldots, s_n$, the formula $s_1 \& s_2 \& \ldots \& s_n \rightarrow s$ is called a theorem, with s the conclusion. When s is a logical consequence of the set $S = \{s_1, \ldots, s_n\}$ we will also say S logically implies or logically entails s, written $S \vdash s$.

It is often convenient to make substitutions when considering compound statements. If s_1 is equivalent to s_2, s_1 may be substituted for s_2 without changing the truth value of a statement or set of sentences containing s_2. Table 4.2 lists some of the important laws of PL. Note that the equal sign as used in the table has the same meaning as $\leftrightarrow$; it also denotes equivalence.

One way to determine the equivalence of two sentences is by using truth tables. For example, to show that $P \rightarrow Q$ is equivalent to $\neg P \vee Q$ and that $P \leftrightarrow Q$ is equivalent to the expression $(P \rightarrow Q) \& (Q \rightarrow P)$, a truth table such as Table 4.3, can be constructed to verify or disprove the equivalences.

TABLE 4.3 TRUTH TABLE FOR EQUIVALENT SENTENCES

P	Q	$\neg P$	$(\neg P \vee Q)$	$(P \rightarrow Q)$	$(Q \rightarrow P)$	$(P \rightarrow Q) \& (Q \rightarrow P)$
true	true	false	true	true	true	true
true	false	false	false	false	true	false
false	true	true	true	true	false	false
false	false	true	true	true	true	true

Inference Rules

The inference rules of PL provide the means to perform logical proofs or deductions. The problem is, given a set of sentences $S = \{s_1, \ldots, s_n\}$ (the premises), prove the truth of s (the conclusion); that is, show that $S \vdash s$. The use of truth tables to do this is a form of semantic proof. Other syntactic methods of inference or deduction are also possible. Such methods do not depend on truth assignments but on syntactic relationships only; that is, it is possible to derive new sentences which are logical consequences of $s_1, \ldots, s_n$ using only syntactic operations. We present a few such rules now which will be referred to often throughout the text.

Modus ponens. From P and $P \rightarrow Q$ infer Q. This is sometimes written as

$$\frac{P \qquad P \rightarrow Q}{Q}$$

For example

given: (joe is a father)
and: (joe is a father) → (joe has a child)
conclude: (joe has a child)

Chain rule. From $P \rightarrow Q$, and $Q \rightarrow R$, infer $P \rightarrow R$. Or

$$\frac{P \rightarrow Q \qquad Q \rightarrow R}{P \rightarrow R}$$

For example,

given: (programmer likes LISP) → (programmer hates COBOL)
and: (programmer hates COBOL) → Programmer likes recursion)
conclude: (programmer likes LISP) → (programmer likes recursion)

Substitution. If s is a valid sentence, s' derived from s by consistent substitution of propositions in s, is also valid. For example, the sentence $P \vee \neg P$ is valid; therefore $Q \vee \neg Q$ is also valid by the substitution rule.

Simplification. From $P \,\&\, Q$ infer P.

Conjunction. From P and from Q, infer $P \,\&\, Q$.

Transposition. From $P \rightarrow Q$, infer $\neg Q \rightarrow \neg P$.

We leave it to the reader to justify the last three rules given above.
We conclude this section with the following definitions.

Formal system. A formal system is a set of axioms S and a set of inference rules L from which new statements can be logically derived. We will sometimes denote a formal system as $<S, L>$ or simply a KB (for knowledge base).

Soundness. Let $<S, L>$ be a formal system. We say the inference procedures L are sound if and only if any statements s that can be derived from $<S, L>$ is a logical consequence of $<S, L>$.

Completeness. Let $<S, L>$ be a formal system. Then the inference procedure L is complete if and only if any sentence s logically implied by $<S, L>$ can be derived using that procedure.

As an example of the above definitions, suppose $S = \{P, P \rightarrow Q\}$ and L is the modus ponens rule. Then $<S, L>$ is a formal system, since Q can be derived from the system. Furthermore, this system is both sound and complete for the reasons given above.

We will see later that a formal system like *resolution* permits us to perform computational reasoning. Clearly, soundness and completeness are desirable properties of such systems. Soundness is important to insure that all derived sentences are true when the assumed set of sentences are true. Completeness is important to guarantee that inconsistencies can be found whenever they exist in a set of sentences.

4.3 SYNTAX AND SEMANTICS FOR FOPL

As was noted in the previous chapter, expressiveness is one of the requirements for any serious representation scheme. It should be possible to accurately represent most, if not all concepts which can be verbalized. PL falls short of this requirement in some important respects. It is too "coarse" to easily describe properties of objects, and it lacks the structure to express relations that exist among two or more entities. Furthermore, PL does not permit us to make generalized statements about classes of similar objects. These are serious limitations when reasoning about real world entities. For example, given the following statements, it should be possible to conclude that John must take the Pascal course.

All students in Computer Science must take Pascal.
John is a Computer Science major.

As stated, it is not possible to conclude in PL that John must take Pascal since the second statement does not occur as part of the first one. To draw the desired conclusion with a valid inference rule, it would be necessary to rewrite the sentences.

FOPL was developed by logicians to extend the expressiveness of PL. It is a generalization of PL that permits reasoning about world objects as relational entities as well as classes or subclasses of objects. This generalization comes from the

introduction of predicates in place of propositions, the use of functions and the use of variables together with variable quantifiers. These concepts are formalized below.

The syntax for FOPL, like PL, is determined by the allowable symbols and rules of combination. The semantics of FOPL are determined by interpretations assigned to predicates, rather than propositions. This means that an interpretation must also assign values to other terms including constants, variables and functions, since predicates may have arguments consisting of any of these terms. Therefore, the arguments of a predicate must be assigned before an interpretation can be made.

Syntax of FOPL

The symbols and rules of combination permitted in FOPL are defined as follows.

Connectives. There are five connective symbols: ¬ (not or negation), & (and or conjunction), V (or or inclusive disjunction, that is, A or B or both A and B), → (implication), ↔ (equivalence or if and only if).

Quantifiers. The two quantifier symbols are ∃ (existential quantification) and ∀ (universal quantification), where (∃x) means for some x or there is an x and (∀x) means for all x. When there is no possibility of confusion, we will omit the parentheses for brevity. Furthermore, when more than one variable is being quantified by the same quantifier such as (∀x) (∀y) (∀z) we abbreviate with a single quantifier and drop the parentheses to get ∀xyz.

Constants. Constants are fixed-value terms that belong to a given domain of discourse. They are denoted by numbers, words, and small letters near the beginning of the alphabet such as a, b, c, 5.3, −21, flight-102, and john.

Variables. Variables are terms that can assume different values over a given domain. They are denoted by words and small letters near the end of the alphabet, such as aircraft-type, individuals, x, y, and z.

Functions. Function symbols denote relations defined on a domain D. They map n elements ($n \geq 0$) to a single element of the domain. Symbols f, g, h, and words such as father-of, or age-of, represent functions. An n place (n-ary) function is written as $f(t_1, t_2, \ldots, t_n)$ where the t_i are terms (constants, variables, or functions) defined over some domain. A 0-ary function is a constant.

Predicates. Predicate symbols denote relations or functional mappings from the elements of a domain D to the values true or false. Capital letters and capitalized words such as P, Q, R, EQUAL, and MARRIED are used to represent predicates. Like functions, predicates may have n ($n \geq 0$) terms for arguments written as $P(t_1, t_2, \ldots, t_n)$, where the terms t_i, $i = 1, 2, \ldots, n$ are defined over some domain. A 0-ary predicate is a proposition, that is, a constant predicate.

Constants, variables, and functions are referred to as *terms*, and predicates are referred to as atomic formulas or *atoms* for short. Furthermore, when we want to refer to an atom or its negation, we often use the word *literal*.

In addition to the above symbols, left and right parentheses, square brackets, braces, and the period are used for punctuation in symbolic expressions.

As an example of the above concepts, suppose we wish to represent the following statements in symbolic form.

E1: All employees earning $1400 or more per year pay taxes.
E2: Some employees are sick today.
E3: No employee earns more than the president.

To represent such expressions in FOPL, we must define abbreviations for the predicates and functions. We might, for example, define the following.

$E(x)$ for x is an employee.
$P(x)$ for x is president.
$i(x)$ for the income of x (lower case denotes a function).
$GE(u,v)$ for u is greater than or equal to v.
$S(x)$ for x is sick today.
$T(x)$ for x pays taxes.

Using the above abbreviations, we can represent E1, E2, and E3 as

$$E1': \forall x\ ((E(x)\ \&\ GE(i(x),1400)) \rightarrow T(x))$$
$$E2': \exists y\ (E(y)\ \&\ S(y))$$
$$E3': \forall x y\ ((E(x)\ \&\ P(y)) \rightarrow \neg GE(i(x),i(y)))$$

In the above, we read E1′ as "for all x if x is an employee and the income of x is greater than or equal to $1400 then x pays taxes." More naturally, we read E1′ as E1, that is as "all employees earning $1400 or more per year pay taxes."

E2′ is read as "there is an employee and the employee is sick today" or "some employee is sick today." E3′ reads "for all x and for all y if x is an employee and y is president, the income of x is *not* greater than or equal to the income of y." Again, more naturally, we read E3′ as, "no employee earns more than the president."

The expressions E1′, E2′, and E3′ are known as well-formed formulas or *wffs* (pronounced woofs) for short. Clearly, all of the wffs used above would be more meaningful if full names were used rather than abbreviations.

Wffs are defined recursively as follows:

An atomic formula is a wff.
If P and Q are wffs, then $\neg P$, $P\ \&\ Q$, $P \vee Q$, $P \rightarrow Q$,

$P \leftrightarrow Q$, $\forall x\ P(x)$, and $\exists x\ P(x)$ are wffs.

Wffs are formed only by applying the above rules a finite number of times.

The above rules state that all wffs are formed from atomic formulas and the proper application of quantifiers and logical connectives.

Some examples of valid wffs are

MAN(john)
PILOT(father-of(bill))
∃xyz ((FATHER(x,y) & FATHER(y,z)) → GRANDFATHER(x,z))
∀x NUMBER(x) → (∃y GREATER-THAN(y,x))
∀x ∃y (P(x) & Q(y)) → (R(a) V Q(b))

Some examples of statements that are *not* wffs are:

∀P P(x) → Q(x)
MAN(john)
father-of(Q(x))
MARRIED(MAN,WOMAN)

The first group of examples above are all wffs since they are properly formed expressions composed of atoms, logical connectives, and valid quantifications. Examples in the second group fail for different reasons. In the first expression, universal quantification is applied to the predicate $P(x)$. This is invalid in FOPL.[1] The second expression is invalid since the term John, a constant, is negated. Recall that predicates, and not terms are negated. The third expression is invalid since it is a function with a predicate argument. The last expression fails since it is a predicate with two predicate arguments.

Semantics for FOPL

When considering specific wffs, we always have in mind some domain D. If not stated explicitly, D will be understood from the context. D is the set of all elements or objects from which fixed assignments are made to constants and from which the domain and range of functions are defined. The arguments of predicates must be terms (constants, variables, or functions). Therefore, the domain of each n-place predicate is also defined over D.

For example, our domain might be all entities that make up the Computer Science Department at the University of Texas. In this case, constants would be professors (Bell, Cooke, Gelfond, and so on), staff (Martha, Pat, Linda, and so on), books, labs, offices, and so forth. The functions we may choose might be

[1] Predicates may be quantified in *second* order predicate logic as indicated in the example, but never in first order logic.

advisor-of(x), lab-capacity(y), dept-grade-average(z), and the predicates MARRIED(x), TENURED(y), COLLABORATE(x,y), to name a few.

When an assignment of values is given to each term and to each predicate symbol in a wff, we say an *interpretation* is given to the wff. Since literals always evaluate to either true or false under an interpretation, the value of any given wff can be determined by referring to a truth table such as Table 4.2 which gives truth values for the subexpressions of the wff.

If the truth values for two different wffs are the same under every interpretation, they are said to be *equivalent*. A predicate (or wff) that has no variables is called a *ground atom*.

When determining the truth value of a compound expression, we must be careful in evaluating predicates that have variable arguments, since they evaluate to true only if they are true for the appropriate value(s) of the variables. For example, the predicate $P(x)$ in $\forall x\ P(x)$, is true only if it is true for every value of x in the domain D. Likewise, the $P(x)$ in $\exists x\ P(x)$ is true only if it is true for at least one value of x in the domain. If the above conditions are not satisfied, the predicate evaluates to false.

Suppose, for example, we want to evaluate the truth value of the expression E, where

$$E: \forall x\ ((A(a,x) \vee B(f(x))) \& C(x)) \rightarrow D(x)$$

In this expression, there are four predicates: A, B, C, and D. The predicate A is a two-place predicate, the first argument being the constant a, and the second argument, a variable x. The predicates B, C, and D are all unary predicates where the argument of B is a function $f(x)$, and the argument of C and D is the variable x.

Since the whole expression E is quantified with the universal quantifier $\forall x$, it will evaluate to true only if it evaluates to true for all x in the domain D. Thus, to complete our example, suppose E is interpreted as follows: Define the domain $D = \{1,2\}$ and from D let the interpretation I assign the following values:

a = 2
f(1) = 2, f(2) = 1
A(2,1) = true, A(2,2) = false
B(1) = true, B(2) = false
C(1) = true, C(2) = false
D(1) = false, D(2) = true

Using a table such as Table 4.3 we can evaluate E as follows:

a. If $x = 1$, $A(2,1)$ evaluates to true, $B(2)$ evaluates to false, and $(A(2,1) \vee B(2))$ evaluates to true. $C(1)$ evaluates to true. Therefore, the expression in

the outer parentheses (the antecedent of E) evaluates to true. Hence, since $D(1)$ evaluates to false, the expression E evaluates to false.

b. In a similar way, if $x = 2$, the expression can be shown to evaluate to true. Consequently, since E is not true for all x, the expression E evaluates to false.

4.4 PROPERTIES OF WFFS

As in the case of PL, the evaluation of complex formulas in FOPL can often be facilitated through the substitution of equivalent formulas. Table 4.3 lists a number of equivalent expressions. In the table F, G and H denote wffs not containing variables and $F[x]$ denotes the wff F which contains the variable x. The equivalences can easily be verified with truth tables such as Table 4.3 and simple arguments for the expressions containing quantifiers. Although Tables 4.4 and 4.3 are similar, there are some notable differences, particularly in the wffs containing quantifiers. For example, attention is called to the last four expressions which govern substitutions involving negated quantifiers and the movement of quantifiers across conjunctive and disjunctive connectives.

We summarize here some definitions which are similar to those of the previous section. A wff is said to be *valid* if it is true under every interpretation. A wff that is false under every interpretation is said to be *inconsistent* (or unsatisfiable). A wff that is not valid (one that is false for some interpretation) is *invalid*. Likewise, a wff that is not inconsistent (one that is true for some interpretation) is *satisfiable*. Again, this means that a valid wff is satisfiable and an inconsistent wff is invalid,

TABLE 4.4. EQUIVALENT LOGICAL EXPRESSIONS

$\neg(\neg F) = F$	(double negation)
$F \& G = G \& F$, $F \vee G = G \vee F$	(commutativity)
$(F \& G) \& H = F \& (G \& H)$,	
$(F \vee G) \vee H = F \vee (G \vee H)$	(associativity)
$F \vee (G \& H) = (F \vee G) \& (F \vee H)$,	
$F \& (G \vee H) = (F \& G) \vee (F \& H)$	(distributivity)
$\neg(F \& G) = \neg F \vee \neg G$,	
$\neg(F \vee G) = \neg F \& \neg G$	(De Morgan's Laws)
$F \rightarrow G = \neg F \vee G$	
$F \leftrightarrow G = (\neg F \vee G) \& (\neg G \vee F)$	
$\forall x\, F[x] \vee G = \forall x\, (F[x] \vee G)$,	
$\exists x\, F[x] \vee G = \exists x\, (F[x] \vee G)$	
$\forall x\, F[x] \& G = \forall x\, (F[x] \& G)$,	
$\exists x\, F[x] \& G = \exists x\, (F[x] \& G)$	
$\neg(\forall x)\, F[x] = \exists x\, (\neg F[x])$.	
$\neg(\exists x)\, F[x] = \forall x\, (\neg F[x])$	
$\forall x\, F[x] \& \forall x\, G[x] = \forall x\, (F[x] \& G[x])$	
$\exists x\, F[x] \vee \exists x\, G[x] = \exists x\, (F[x] \vee G[x])$	

but the respective converse statements do not hold. Finally, we say that a wff Q is a *logical consequence* of the wffs $P_1, P_2, \ldots, P_n$ if and only if whenever P_1 & P_2 & . . . & P_n is true under an interpretation, Q is also true.

To illustrate some of these concepts, consider the following examples:

a. P & $\neg P$ is inconsistent and $P \vee \neg P$ is valid since the first is false under every interpretation and the second is true under every interpretation.

b. From the two wffs

$$\text{CLEVER(bill) and}$$
$$\forall x\ \text{CLEVER}(x) \rightarrow \text{SUCCEED}(x)$$

we can show that SUCCEED(bill) is a logical consequence. Thus, assume that both

$$\text{CLEVER(bill) and}$$
$$\forall x\ \text{CLEVER}(x) \rightarrow \text{SUCCEED}(x)$$

are true under an interpretation. Then

$$\text{CLEVER(bill)} \rightarrow \text{SUCCEED(bill)}$$

is certainly true since the wff was assumed to be true for all x, including x = bill. But,

$$\text{CLEVER(bill)} \rightarrow \text{SUCCEED(bill)}$$
$$= \neg\text{CLEVER(bill)} \vee \text{SUCCEED(bill)}$$

are equivalent and, since CLEVER(bill) is true, ¬CLEVER(bill) is false and, therefore, SUCCEED(bill) must be true. Thus, we conclude SUCCEED-(bill) is a logical consequence of

$$\text{CLEVER(bill) and } \forall x\ \text{CLEVER}(x) \rightarrow \text{SUCCEED}(x).$$

Suppose the wff $F[x]$ contains the variable x. We say x is *bound* if it follows or is within the scope of a quantifier naming the variable. If a variable is not bound, it is said to be free. For example, in the expression $\forall x\ (P(x) \rightarrow Q(x,y))$, x is bound, but y is free since every occurrence of x follows the quantifier and y is not within the scope of any quantifier. Clearly, an expression can be evaluated only when all the variables in that expression are bound. Therefore, we shall require that all wffs contain only bound variables. We will also call such expressions a sentence.

We conclude this section with a few more definitions. Given wffs F_1, F_2,

. . . , F_n each possibly consisting of the disjunction of literals only, we say F_1 & F_2 & . . . & F_n is in *conjunctive normal form* (CNF). On the other hand if each F_i, $i = 1, \ldots, n$ consists only of the conjunction of literals, we say $F_1 \vee F_2 \vee \ldots \vee F_n$ is in *disjunctive normal form* (DNF). For example, the wffs $(\neg P \vee Q \vee R) \& (\neg P \vee \neg Q) \& \neg R$ and $(P \& Q \& R) \vee (Q \& R) \vee P$ are in conjunctive and disjunctive normal forms respectively. It can be shown that *any* wff can be transformed into either normal form.

4.5 CONVERSION TO CLAUSAL FORM

As noted earlier, we are interested in mechanical inference by programs using symbolic FOPL expressions. One method we shall examine is called *resolution*. It requires that all statements be converted into a normalized clausal form. We define a *clause* as the disjunction of a number of literals. A *ground clause* is one in which no variables occur in the expression. A *Horn clause* is a clause with at most one positive literal.

To transform a sentence into clausal form requires the following steps:

eliminate all implication and equivalence symbols,

move negation symbols into individual atoms,

rename variables if necessary so that all remaining quantifiers have different variable assignments,

replace existentially quantified variables with special functions and eliminate the corresponding quantifiers,

drop all universal quantifiers and put the remaining expression into CNF (disjunctions are moved down to literals), and

drop all conjunction symbols writing each clause previously connected by the conjunctions on a separate line.

These steps are described in more detail below. But first, we describe the process of eliminating the existential quantifiers through a substitution process. This process requires that all such variables be replaced by something called Skolem functions, arbitrary functions which can always assume a correct value required of an existentially quantified variable.

For simplicity in what follows, assume that all quantifiers have been properly moved to the left side of the expression, and each quantifies a different variable. Skolemization, the replacement of existentially quantified variables with Skolem functions and deletion of the respective quantifiers, is then accomplished as follows:

1. If the first (leftmost) quantifier in an expression is an existential quantifier, replace all occurrences of the variable it quantifies with an arbitrary constant not appearing elsewhere in the expression and delete the quantifier. This same procedure

should be followed for all other existential quantifiers not preceded by a universal quantifier, in each case, using different constant symbols in the substitution.

2. For each existential quantifier that is preceded by one or more universal quantifiers (is within the scope of one or more universal quantifiers), replace all occurrences of the existentially quantified variable by a function symbol not appearing elsewhere in the expression. The arguments assigned to the function should match all the variables appearing in each universal quantifier which precedes the existential quantifier. This existential quantifier should then be deleted. The same process should be repeated for each remaining existential quantifier using a different function symbol and choosing function arguments that correspond to all universally quantified variables that precede the existentially quantified variable being replaced.

An example will help to clarify this process. Given the expression

$$\exists u\ \forall v\ \forall x\ \exists y\ P(f(u), v, x, y) \rightarrow Q(u,v,y)$$

the Skolem form is determined as

$$\forall v\ \forall x\ P(f(a),v,x,g(v,x)) \rightarrow Q(a,v,g(v,x)).$$

In making the substitutions, it should be noted that the variable u appearing after the first existential quantifier has been replaced in the second expression by the arbitrary constant a. This constant did not appear elsewhere in the first expression. The variable y has been replaced by the function symbol g having the variables v and x as arguments, since both of these variables are universally quantified to the left of the existential quantifier for y. Replacement of y by an arbitrary function with arguments v and x is justified on the basis that y, following v and x, may be functionally dependent on them and, if so, the arbitrary function g can account for this dependency. The complete procedure can now be given to convert any FOPL sentence into clausal form.

Clausal Conversion Procedure

Step 1. Eliminate all implication and equivalency connectives (use $\neg P \vee Q$ in place of $P \rightarrow Q$ and $(\neg P \vee Q) \,\&\, (\neg Q \vee P)$ in place of $P \leftrightarrow Q$.

Step 2. Move all negations in to immediately precede an atom (use P in place of $\neg(\neg P)$, and DeMorgan's laws, $\exists x\ \neg F[x]$ in place of $\neg(\forall x)\ F[x]$ and $\forall x\ \neg F[x]$ in place of $\neg(\exists x)\ F[x]$).

Step 3. Rename variables, if necessary, so that all quantifiers have different variable assignments; that is, rename variables so that variables bound by one quantifier are not the same as variables bound by a different quantifier. For example, in the expression $\forall x\ (P(x) \rightarrow (\exists x\ (Q(x))))$ rename the second "dummy" variable x which is bound by the existential quantifier to be a different variable, say y, to give $\forall x\ (P(x) \rightarrow (\exists y\ Q(y)))$.

Step 4. Skolemize by replacing all existentially quantified variables with Skolem functions as described above, and deleting the corresponding existential quantifiers.

Step 5. Move all universal quantifiers to the left of the expression and put the expression on the right into CNF.

Step 6. Eliminate all universal quantifiers and conjunctions since they are retained implicitly. The resulting expressions (the expressions previously connected by the conjunctions) are clauses and the set of such expressions is said to be in clausal form.

As an example of this process, let us convert the expression

$$\exists x\, \forall y\, (\forall z\, P(f(x),y,z) \rightarrow (\exists u\, Q(x,u)\ \&\ \exists v\, R(y,v)))$$

into clausal form. We have after application of step 1

$$\exists x\, \forall y\, \neg(\forall z)\, P(f(x),y,z) \vee (\exists u\, Q(x,u)\ \&\ (\exists v)\, R(y,v)))).$$

After application of step 2 we obtain

$$\exists x\, \forall y\, (\exists z\, \neg P(f(x),y,z) \vee (\exists u\, Q(x,u)\ \&\ (\exists v)\, R(y,v)))).$$

After application of step 4 (step 3 is not required)

$$\forall y\, \neg P(f(a),y,g(y)) \vee (Q(a,h(y))\ \&\ R(y,l(y)))).$$

After application of step 5 the result is

$$\forall y\, ((\neg P(f(a),y,g(y)) \vee Q(a,h(y))\ \&\ \neg P(f(a),y,g(y)) \vee R(y,l(y)))).$$

Finally, after application of step 6 we obtain the clausal form

$$\neg P(f(a),y,g(y)) \vee Q(a,h(y))$$
$$\neg P(f(a),y,g(y) \vee R(y,l(y))$$

The last two clauses of our final form are understood to be universally quantified in the variable y and to have the conjunction symbol connecting them.

It should be noted that the set of clauses produced by the above process are not *equivalent* to the original expression, but *satisfiability* is retained. That is, the set of clauses are satisfiable if and only if the original sentence is satisfiable.

Having now labored through the tedious steps above, we point out that it is often possible to write down statements directly in clausal form without working through the above process step-by-step. We illustrate how this may be done in Section 4.7 when we create a sample knowledge base.

4.6 INFERENCE RULES

Like PL, a key inference rule in FOPL is modus ponens. From the assertion "Leo is a lion" and the implication "all lions are ferocious" we can conclude that Leo is ferocious. Written in symbolic form we have

assertion: LION(leo)
implication: $\forall x$ LION(x) $\rightarrow$ FEROCIOUS(x)
conclusion: FEROCIOUS(leo)

In general, if a has property P and all objects that have property P also have property Q, we conclude that a has property Q.

$$\frac{P(a) \qquad \forall x\, P(x) \rightarrow Q(x)}{Q(a)}$$

Note that in concluding $Q(a)$, a substitution of a for x was necessary. This was possible, of course, since the implication $P(x) \rightarrow Q(x)$ is assumed true for all x, and in particular for $x = a$. Substitutions are an essential part of the inference process. When properly applied, they permit simplifications or the reduction of expressions through the cancellation of complementary literals. We say that two literals are *complementary* if they are identical but of opposite sign; that is, P and $\neg P$ are complementary.

A *substitution* is defined as a set of pairs t_i and v_i where v_i are distinct variables and t_i are terms not containing the v_i. The t_i replace or are substituted for the corresponding v_i in any expression for which the substitution is applied. A set of substitutions $\{t_1/v_1, t_2/v_2, \ldots, t_n/v_n\}$ where $n \geq 1$ applied to an expression will be denoted by Greek letters α, β, and δ. For example, if $\beta = \{a/x, g(b)/y\}$, then applying β to the clause $C = P(x,y) \vee Q(x,f(y))$ we obtain $C' = C\beta = P(a,g(b)) \vee Q(a,f(g(b)))$.

Unification

Any substitution that makes two or more expressions equal is called a *unifier* for the expressions. Applying a substitution to an expression E produces an *instance* E' of E where $E' = E\beta$. Given two expressions that are unifiable, such as expressions C_1 and C_2 with a unifer β with $C_1\beta = C_2$, we say that β is a *most general unifer* (mgu) if any other unifer α is an instance of β. For example two unifiers for the literals $P(u,b,v)$ and $P(a,x,y)$ are $\alpha = \{a/u,b/x,v/y\}$ and $\beta = \{a/u,b/x,c/v,c/y\}$. The former is an mgu whereas the latter is not since it is an *instance* of the former.

Unification can sometimes be applied to literals within the same single clause. When an mgu exists such that two or more literals within a clause are unified, the clause remaining after deletion of all but one of the unified literals is called a

factor of the original clause. Thus, given the clause $C = P(x) \vee Q(x,y) \vee P(f(z))$ the factor $C' = C\beta = P(f(z)) \vee Q(f(z),y)$ is obtained where $\beta = \{f(z)/x\}$.

Let **S** be a set of expressions. We define the *disagreement set* of **S** as the set obtained by comparing each symbol of all expressions in **S** from left to right and extracting from **S** the subexpressions whose first symbols do not agree. For example, let $\mathbf{S} = \{P(f(x),g(y),a), P(f(x),z,a), P(f(x),b,h(u))\}$. For the set **S**, the disagreement set is $\{g(y),a,b,z,h(u)\}$. We can now state a unification algorithm which returns the mgu for a given set of expressions **S**.

Unification algorithm:

1. Set $k = 0$ and $\sigma_k = e$ (the empty set).
2. If the set $\mathbf{S}\sigma_k$ is a singleton, then stop; σ_k is an mgu of **S**. Otherwise, find the disagreement set D_k of $\mathbf{S}\sigma_k$.
3. If there is a variable v and term t in D_k such that v does not occur in t, put $\sigma_{k+1} = \sigma_k\{t/v\}$, set $k = k + 1$, and return to step 2. Otherwise, stop. **S** is not unifiable.

4.7 THE RESOLUTION PRINCIPLE

We are now ready to consider the resolution principle, a syntactic inference procedure which, when applied to a set of clauses, determines if the set is unsatisfiable. This procedure is similar to the process of obtaining a proof by contradiction. For example, suppose we have the set of clauses (axioms) $C_1, C_2, \ldots, C_n$ and we wish to deduce or prove the clause D, that is, to show that D is a logical consequence of $C_1 \,\&\, C_2 \,\&\, \ldots \,\&\, C_n$. First, we negate D and add $\neg D$ to the set of clauses $C_1, C_2, \ldots, C_n$. Then, using resolution together with factoring, we can show that the set is unsatisfiable by deducing a contradiction. Such a proof is called a proof by refutation which, if successful, yields the empty clause denoted by [ ].[2] Resolution with factoring is *complete* in the sense that it will always generate the empty clause from a set of unsatisfiable clauses.

Resolution is very simple. Given two clauses C_1 and C_2 with no variables in common, if there is a literal l_1 in C_1 which is a complement of a literal l_2 in C_2, both l_1 and l_2 are deleted and a disjuncted C is formed from the remaining reduced clauses. The new clause C is called the *resolvent* of C_1 and C_2. Resolution is the process of generating these resolvents from a set of clauses. For example, to resolve the two clauses

$$(\neg P \vee Q) \text{ and } (\neg Q \vee R)$$

[2] The empty clause [ ] is always false since no interpretation can satisfy it. It is derived from combining contradictory clauses such as P and $\neg P$.

we write

$$\frac{\neg P \vee Q,\ \neg Q \vee R}{\neg P \vee R}$$

Several types of resolution are possible depending on the number and types of parents. We define a few of these types below.

Binary resolution. Two clauses having complementary literals are combined as disjuncts to produce a single clause after deleting the complementary literals. For example, the binary resolvent of

$$\neg P(x,a) \vee Q(x) \quad \text{and} \quad \neg Q(b) \vee R(x)$$

is just

$$\neg P(b,a) \vee R(b).$$

The substitution $\{b/x\}$ was made in the two parent clauses to produce the complementary literals $Q(b)$ and $\neg Q(b)$ which were then deleted from the disjunction of the two parent clauses.

Unit resulting (UR) resolution. A number of clauses are resolved simultaneously to produce a unit clause. All except one of the clauses are unit clauses, and that one clause has exactly one more literal than the total number of unit clauses. For example, resolving the set

$$\{\neg \text{MARRIED}(x,y) \vee \neg \text{MOTHER}(x,z) \vee \text{FATHER}(y,z),\ \text{MARRIED}(\text{sue},\text{joe}),\ \neg \text{FATHER}(\text{joe},\text{bill})\}$$

where the substitution $\beta = \{\text{sue}/x,\ \text{joe}/y,\ \text{bill}/z\}$ is used, results in the unit clause $\neg \text{MOTHER}(\text{sue},\text{bill})$.

Linear resolution. When each resolved clause C_i is a parent to the clause C_{i+1} $(i = 1, 2, \ldots, n-1)$ the process is called linear resolution. For example, given a set S of clauses with $C_0 \subseteq S$, C_n is derived by a sequence of resolutions, C_0 with some clause B_0 to get C_1, then C_1 with some clause B_1 to get C_2, and so on until C_n has been derived.

Linear input resolution. If one of the parents in linear resolution is always from the original set of clauses (the B_i), we have linear input resolution. For example, given the set of clauses $S = \{P \vee Q,\ \neg P \vee Q,\ P \vee \neg Q,\ \neg P \vee \neg Q\}$ let $C_0 = (P \vee Q)$. Choosing $B_0 = \neg P \vee Q$ from the set S and resolving this with C_0 we obtain the resolvent $Q = C_1$. B_1 must now be chosen from S and the resolvent of C_1 and B_1 becomes C_2 and so on.

Unification and resolution give us one approach to the problem of mechanical inference or automated reasoning, but without some further refinements, resolution

can be intolerably inefficient. Randomly resolving clauses in a large set can result in inefficient or even impossible proofs. Typically, the curse of combinatorial explosion occurs. So methods which constrain the search in some way must be used.

When attempting a proof by resolution, one ideally would like a minimally unsatisfiable set of clauses which includes the conjectured clause. A *minimally unsatisfiable set* is one which is satisfiable when any member of the set is omitted. The reason for this choice is that irrelevant clauses which are not needed in the proof but which participate are unnecessary resolutions. They contribute nothing toward the proof. Indeed, they can sidetrack the search direction resulting in a dead end and loss of resources. Of course, the set must be unsatisfiable otherwise a proof is impossible.

A minimally unsatisfiable set is ideal in the sense that all clauses are essential and no others are needed. Thus, if we wish to prove B, we would like to do so with a set of clauses $S = \{A_1, A_2, \ldots, A_k\}$ which become minimally unsatisfiable with the addition of $\neg B$.

Choosing the order in which clauses are resolved is known as a search strategy. While there are many such strategies now available, we define only one of the more important ones, the set-of-support strategy. This strategy separates a set which is unsatisfiable into subsets, one of which is satisfiable.

Set-of-support strategy. Let S be an unsatisfiable set of clauses and T be a subset of S. Then T is a set-of-support for S if $S - T$ is satisfiable. A set-of-support resolution is a resolution of two clauses not both from $S - T$. This essentially means that given an unsatisfiable set $\{A_1, \ldots, A_k\}$, resolution should not be performed directly among the A_i as noted above.

Example of Resolution

The example we present here is one to which all AI students should be exposed at some point in their studies. It is the famous "monkey and bananas problem," another one of those complex real life problems solvable with AI techniques. We envision a room containing a monkey, a chair, and some bananas that have been hung from the center of the ceiling, out of reach from the monkey. If the monkey is clever enough, he can reach the bananas by placing the chair directly below them and climbing on top of the chair. The problem is to use FOPL to represent this monkey-banana world and, using resolution, prove the monkey can reach the bananas.

In creating a knowledge base, it is essential first to identify all relevant objects which will play some role in the anticipated inferences. Where possible, irrelevant objects should be omitted, but never at the risk of incompleteness. For example, in the current problem, the monkey, bananas, and chair are essential. Also needed is some reference object such as the floor or ceiling to establish the height relationship between monkey and bananas. Other objects such as windows, walls or doors are not relevant.

The next step is to establish important properties of objects, relations between

them, and any assertions likely to be needed. These include such facts as the chair is tall enough to raise the monkey within reach of the bananas, the monkey is dexterous, the chair can be moved under the bananas, and so on. Again, all important properties, relations, and assertions should be included and irrelevant ones omitted. Otherwise, unnecessary inference steps may be taken.

The important factors for our problem are described below, and all items needed for the actual knowledge base are listed as axioms. These are the essential facts and rules. Although not explicitly indicated, all variables are universally quantified.

Relevant factors for the problem

CONSTANTS

{floor, chair, bananas, monkey}

VARIABLES

{x, y, z}

PREDICATES

```
{can_reach(x,y)      ; x can reach y
dexterous (x)        ; x is a dexterous animal
close(x,y)           ; x is close to y
get_on(x,y)          ; x can get on y
under(x,y)           ; x is under y
tall(x)              ; x is tall
in_room(x)           ; x is in the room
can_move(x,y,z)      ; x can move y near z
can_climb(x,y)}      ; x can climb onto y
```

AXIOMS

```
{in_room(bananas)
in_room(chair)
in_room(monkey)
dexterous(monkey)
tall(chair)
 close(bananas,floor)
can_move(monkey,chair,bananas)
can_climb(monkey,chair)
(dexterous(x) & close(x,y) → can-reach(x,y)
((get_on(x,y) & under(y,bananas) & tall(y) →
   close(x,bananas))
((in_room(x) & in_room(y) & in_room(z) & can_move(x,y,z))
    → close(z,floor) V under(y,z))
(can_climb(x,y) → get_on(x,y))}
```

Using the above axioms, a knowledge base can be written down directly in the required clausal form. All that is needed to make the necessary substitutions are the equivalences

$$P \rightarrow Q = \neg P \vee Q$$

and De Morgan's laws. To relate the clauses to a LISP program, one may prefer to think of each clause as being a list of items. For example, number 9, below, would be written as

```
(or (~can_climb(?x ?y) get_on(?x ?y))
```

where ?*x* and ?*y* denote variables.

Note that clause 13 is not one of the original axioms. It has been added for the proof as required in refutation resolution proofs.

Clausal form of knowledge base

1. in_room(monkey)
2. in_room(bananas)
3. in_room(chair)
4. tall(chair)
5. dexterous(monkey)
6. can_move(monkey, chair, bananas)
7. can_climb(monkey,chair)
8. ~close(bananas,floor)
9. ~can_climb(x,y) V get_on(x,y)
10. ~dexterous(x) V ~close(x,y) V can_reach(x,y)
11. ~get_on(x,y) V ~under(y,bananas) V ~tall(y) V close(x,bananas)
12. ~in_room(x) V ~in_room(y) V ~in_room(z) V ~can_move(x,y,z) V close(y,floor) V under(y,z)
13. ~can_reach(monkey,bananas)

Resolution proof. A proof that the monkey can reach the bananas is summarized below. As can be seen, this is a refutation proof where the statement to be proved (can_reach(monkey,bananas)) has been negated and added to the knowledge base (number 13). The proof then follows when a contradiction is found (see number 23, below).

14. ~can_move(monkey,chair,bananas) V close(bananas,floor) V under (chair,bananas) ; 14 is a resolvent of 1,2,3 and 12 with substitution (monkey/*x*, chair *y*, bananas/*z*)

15.	close(bananas,floor) V under (chair,bananas)	; this is a resolvent of 6 and 14
16.	under(chair,bananas)	; this is a resolvent of 8 and 15
17.	~get_on(x,chair) V ~tall(chair) V close(x,bananas)	; this is a resolvent of 11 and 16 with substitution {chair/y}
18.	~get_on(x,chair) V close(x,bananas)	; a resolvent of 4 and 17
19.	get_on(monkey,chair)	; a resolvent of 7 and 9
20.	close(monkey,bananas)	; a resolvent of 18 and 19 with substitution {monkey/x}
21.	~close(monkey,y) V can_reach (monkey,y)	; a resolvent of 10 and 5 with substitution {monkey/x}
22.	reach(monkey,bananas)	; a resolvent of 20 and 21 with substitution {bananas/y}
23.	[]	; a resolvent of 13 and 22

In performing the above proof, no particular strategy was followed. Clearly, however, good choices were made in selecting parent clauses for resolution. Otherwise, many unnecessary steps may have been taken before completing the proof. Different forms of resolution were completed in steps 14 through 23. One of the exercises requires that the types of resolutions used be identified.

The Monkey-Banana Problem in PROLOG

Prolog was introduced in the previous chapter. It is a logic programming language that is based on the resolution principle. The refutation proof strategy used in PROLOG is resolution by selective linear with definite clauses or SLD resolution. This is just a form of linear input resolution using definite Horn clauses (clauses with exactly one positive literal). In finding a resolution proof, PROLOG searches a data base of clauses in an exhaustive manner (guided by the SLD strategy) until a chain of unifications have been found to produce a proof.

Next, we present a PROLOG program for the monkey-banana problem to illustrate the ease with which many logic programs may be formulated.

```
% Constants:
% {floor, chair, bananas, monkey}

% Variables:
% {X, Y, Z}

% Predicates:
% {can-reach(X,Y)        ; X can reach Y
%  dexterous(X)          ; X is a dexterous animal
%  close(X,Y)            ; X is close to Y
```

```
% get-on(X,Y)          ; X can get on Y
% under(X,Y)           ; X is under Y
% tall(X)              ; X is tall
% in-room(X)           ; X is in the room
% can-move(X,Y,Z)      ; X can move Y near Z
% can-climb(X,Y)}      ; X can climb onto Y

% Axioms :

in-room(bananas).

in-room(chair).

in-room(monkey).

dexterous(monkey).

tall(chair).

can-move(monkey, chair, bananas).

can-climb(monkey, chair).

can-reach(X,Y) :-

        dexterous(X), close(X,Y).

close(X,Z) :-

        get-on(X,Y),

        under(Y,Z),

        tall(Y).

get-on(X,Y) :-

        can-climb(X,Y).

Under(Y,Z) :-

        in-room(X),

        in-room(Y),

        in-room(Z),

        can-move(X,Y,Z).
```

This completes the data base of facts and rules required. Now we can pose various queries to our theorem proving system.

```
| ?- can-reach(X,Y).

X = monkey,

Y = bananas

| ?- can-reach(X,bananas).

X = monkey

| ?- can-reach(monkey,Y).

Y = bananas

| ?- can-reach(monkey,bananas).

yes

| ?- can-reach(lion,bananas).

no

| ?- can-reach(monkey,apple).

no
```

4.8 NONDEDUCTIVE INFERENCE METHODS

In this section we consider three nondeductive forms of inferencing. These are not valid forms of inferencing, but they are nevertheless very important. We use all three methods often in every day activities where we draw conclusions and make decisions. The three methods we consider here are abduction, induction, and analogical inference.

Abductive Inference

Abductive inference is based on the use of known causal knowledge to explain or justify a (possibly invalid) conclusion. Given the truth of proposition Q and the implication $P \rightarrow Q$, conclude P. For example, people who have had too much to drink tend to stagger when they walk. Therefore, it is not unreasonable to conclude that a person who is staggering is drunk even though this may be an incorrect

conclusion. People may stagger when they walk for other reasons, including dizziness from twirling in circles or from some physical problem.

We may represent abductive inference with the following, where the c over the implication arrow is meant to imply a possible causal relationship.

$$\begin{array}{ll} \text{assertion} & Q \\ \text{implication} & P \overset{c}{\rightarrow} Q \\ \text{conclusion} & \overline{P} \end{array}$$

Abductive inference is useful when known causal relations are likely and deductive inferencing is not possible for lack of facts.

Inductive Inference

Inductive inferencing is based on the assumption that a recurring pattern, observed for some event or entity, implies that the pattern is true for all entities in the class. Given instances $P(a_1)$, $P(a_2)$, . . . , $P(a_k)$, conclude that $\forall x\, P(x)$. More generally, given $P(a_1) \rightarrow Q(b_1)$, $P(a_2) \rightarrow Q(b_2)$, . . . , $P(a_k) \rightarrow Q(b_k)$, conclude $\forall x,y\, P(x) \rightarrow Q(y)$.

We often make this form of generalization after observing only a few instances of a situation. It is known as the *inductive leap*. For example, after seeing a few white swans, we incorrectly infer that all swans are white (a type of Australian swan is black), or we conclude that all Irishmen are stubborn after discussions with only a few.

We can represent inductive inference using the following description:

$$\frac{P(a_1), \ldots, P(a_k)}{\forall x\, P(x)}$$

Inductive inference, of course, is not a valid form of inference, since it is not usually the case that all objects of a class can be verified as having a particular property. Even so, this is an important and commonly used form of inference.

Analogical Inference

Analogical inference is a form of experiential inference. Situations or entities which are alike in some respects tend to be similar in other respects. Thus, when we find that situation (object) A is related in certain ways to B, and A' is similar in some context to A, we conclude that B' has a similar relation to A' in this context. For example, to solve a problem with three equations in three unknowns, we try to extend the methods we know in solving two equations in two unknowns.

Analogical inference appears to be based on the use of a combination of three other methods of inference, abductive, deductive and inductive. We depict this form of inference with the following description, where the r above the implication symbol means is related to.

$$\frac{P' \rightarrow Q}{P'' \rightarrow Q'}$$

Analogical inference, like abductive and inductive is a useful but invalid form of commonsense inference.

4.9 REPRESENTATIONS USING RULES

Rules can be considered a subset of predicate logic. They have become a popular representation scheme for expert systems (also called rule-based systems). They were first used in the General Problem Solver system in the early 1970s (Newell and Simon, 1972).

Rules have two component parts: a left-hand side (LHS) referred to as the antecedent, premise, condition, or situation, and a right-hand side (RHS) known as the consequent, conclusion, action, or response. The LHS is also known as the if part and the RHS as the then part of the rule. Some rules also include an else part. Examples of rules which might be used in expert systems are given below.

IF: The temperature is greater than 95 degrees C.
THEN: Open the relief valve.

IF: The patient has a high temperature,
and the stain is gram-positive,
and the patient has a sore throat,
THEN: The organism is streptococcus.

IF: The lights do not come on,
and the engine does not turn over,
THEN: The battery is dead or the cable is loose.

IF: $A \ \& \ B \ \& \ C$
THEN: D
$A \ \& \ B \ \& \ (C \vee D) \rightarrow D$

The simplest form of a rule-based production system consists of three parts, a knowledge base (KB) consisting of a set of rules (as few as 50 or as many as several thousand rules may be required in an expert system), a working memory, and a rule interpreter or inference engine. The interpreter inspects the LHS of each rule in the KB until one is found which matches the contents of working memory. This causes the rule to be activated or to "fire" in which case the contents of working memory are replaced by the RHS of the rule. The process continues by scanning the next rules in sequence or restarting at the beginning of the knowledge base.

INTERNAL FORM

```
RULE047
    Premise:  (($and (same cntxt site blood)
                     (notdefinite contxt ident)
                     (same cntxt morph rod)
                     (same cntxt burn t))
    Action:   (conclude cntxt ident pseudonomas 0.4))
```

ENGLISH TRANSLATION

IF: 1) The site of the culture is blood, and
2) The identity of the organism is not known with certainty, and
3) The stain of the organism is gramneg, and
4) The morphology of the organism is rod, and
5) The patient has been seriously burned

THEN: There is weakly suggestive evidence (0.4) that the identity of the organism is pseudonomas.

Figure 4.1 A rule from the MYCIN system.

Each rule represents a chunk of knowledge as a conditional statement, and each invocation of the rules as a sequence of actions. This is essentially an inference process using the chain rule as described in Section 4.2. Although there is no established syntax for rule-based systems, the most commonly used form permits a LHS consisting of a conjunction of several conditions and a single RHS action term.

An example of a rule used in the MYCIN[3] expert system (Shortliffe, 1976) is illustrated in Figure 4.1.

In RULE047, the quantity 0.4 is known as a confidence factor (CF). Confidence factors range from -1.0 (complete disbelief) to 1.0 (certain belief). They provide a measure of the experts' confidence that a rule is applicable or holds when the conditions in the LHS have all been satisfied.

We will see further examples of rules and confidence factors in later chapters.

4.10 SUMMARY

We have considered propositional and first order predicate logics in this chapter as knowledge representation schemes. We learned that while PL has a sound theoretical foundation, it is not expressive enough for many practical problems. FOPL, on the

[3] MYCIN was one of the earliest expert systems. It was developed at Stanford University in the mid-1970s to demonstrate that a system could successfully perform diagnoses of patients having infectious blood diseases.

other hand, provides a theoretically sound basis and permits a great latitude of expressiveness. In FOPL one can easily code object descriptions and relations among objects as well as general assertions about classes of similar objects. The increased generality comes from the joining of predicates, functions, variables, and quantifiers. Perhaps the most difficult aspect in using FOPL is choosing appropriate functions and predicates for a given class of problems.

Both the syntax and semantics of PL and FOPL were defined and examples given. Equivalent expressions were presented and the use of truth tables was illustrated to determine the meaning of complex formulas. Rules of inference were also presented, providing the means to derive conclusions from a basic set of facts or axioms. Three important syntactic inference methods were defined: modus ponens, chain rule and resolution. These rules may be summarized as follows.

MODUS PONENS	CHAIN RULE	RESOLUTION
$\dfrac{P \quad P \rightarrow Q}{Q}$	$\dfrac{P \rightarrow Q \quad Q \rightarrow R}{P \rightarrow R}$	$\dfrac{P \vee \neg Q, \; Q \vee R}{P \vee R}$

A detailed procedure was given to convert any complex set of formulas to clausal normal forms. To perform automated inference or theorem proving, the resolution method requires that the set of axioms and the conjecture to be proved be in clausal form. Resolution is important because it provides the means to mechanically derive conclusions that are valid. Programs using resolution methods have been developed for numerous systems with varying degrees of success, and the language PROLOG is based on the use of such resolution proofs. FOPL has without question become one of the leading methods for knowledge representation.

In addition to valid forms of inference based on deductive methods, three invalid, but useful types of inferencing were also presented, abductive, inductive, and analogical.

Finally, rules, a subset of FOPL, were described as a popular representation scheme. As will be seen in Chapter 15, many expert systems use rules for their knowledge bases. Rules provide a convenient means of incrementally building a knowledge base in an easily understood language.

EXERCISES

4.1 Construct a truth table for the expression $(A \; \& \; (A \vee B))$. What single term is this expression equivalent to?

4.2 Prove the following rules from Section 4.2.

(a) Simplification: From $P \; \& \; Q$, infer P

(b) Conjunction: From P and Q, infer $P \; \& \; Q$

(c) Transposition: From $P \rightarrow Q$, infer $\neg Q \rightarrow \neg P$

4.3 Given the following PL expressions, place parentheses in the appropriate places to form fully abbreviated wffs.

(a) $\neg P \vee Q \,\&\, R \rightarrow S \rightarrow U \,\&\, Q$

(b) $\neg P \,\&\, Q \vee P \leftrightarrow U \rightarrow \neg R$

(c) $Q \vee P \vee \neg R \,\&\, S \rightarrow \neg U \,\&\, P \leftrightarrow R$

4.4 Translate the following axioms into predicate calculus wffs. For example, A_1 below could be given as

$$\forall x,y,z \quad \text{CONNECTED}(x,y,z) \,\&\, \text{Bikesok}(z) \rightarrow \text{GETTO}(x,y)$$

A1. If town x is connected to town y by highway z, and bicycles are allowed on z, you can get to y from x by bike.

A2. If town x is connected to y by z, y is connected to x by z.

A3. If you can get to y from x, and you can get to z from y, you can get to z from x.

A4. Town-A is connected to Town-B by Road-1.

A5. Town-B is connected to Town-C by Road-2.

4.5 Convert axioms A1-A5 from Problem 4.4 to clausal form and write axioms A6-A13, below, directly into clausal form.

A6. Town-A is connected to Town-C by Road-3.

A7. Town-D is connected to Town-E by Road-4.

A8. Town-D is connected to Town-B by Road-5.

A9-A11. Bikes are allowed on Road-3, Road-4, and Road-5.

A12. Bikes are always allowed on either Road-2 or Road-1 (each day it may be different but one road is always possible).

A13. Town-A and Town-E are not connected by Road-3.

4.6 Use the clauses from Problem 4.5 to answer the following questions:

(a) What can be deduced from clauses 2 and 13?

(b) Can GETTO(Town-D, Town-B) be deduced from the above clauses?

4.7 Find the meaning of the statement

$$(\neg P \vee Q) \,\&\, R \rightarrow S \vee (\neg \,\&\, Q)$$

for each of the interpretations given below.

(a) I_1: P is true, Q is true, R is false, S is true.

(b) I_2: P is true, Q is false, R is true, S is true.

4.8 Transform each of the following sentences into disjunctive normal form.

(a) $\neg(P \,\&\, Q) \,\&\, (P \vee Q)$

(b) $\neg(P \vee \neg Q) \,\&\, (R \rightarrow S)$

(c) $P \rightarrow ((Q \,\&\, R) \leftrightarrow S)$

4.9 Transform each of the following sentences into conjunctive normal form.

(a) $(P \rightarrow Q) \rightarrow R$

(b) $P \vee (\neg P \,\&\, Q \,\&\, R)$

(c) $(\neg P \,\&\, Q) \vee (P \,\&\, \neg Q) \,\&\, S$

4.10 Determine whether each of the following sentences is

(a) satisfiable

(b) contradictory

(c) valid

S_1: $(P \& Q) \vee \neg(P \& Q)$ S_2: $(P \vee Q) \rightarrow (P \& Q)$
S_3: $(P \& Q) \rightarrow R \vee \neg Q$ S_4: $(P \vee Q) \& (P \vee \neg Q) \vee P$
S_5: $P \rightarrow Q \rightarrow \neg P$ S_6: $P \vee Q \& \neg P \vee \neg Q \& P$

4.11 Given the following wffs $P \rightarrow Q$, $\neg Q$, and $\neg P$, show that $\neg P$ is a logical consequene of the two preceding wffs:
(a) Using a truth table
(b) Using Theorem 4.1.

4.12 Given formulas S_1 and S_2 below, show that $Q(a)$ is a logical consequence of the two.

$$S_1: (\forall x)(P(x) \rightarrow Q(x)) \qquad S_2: P(a)$$

4.13 Transform the following formula to prenex normal form

$$\forall xy\ (\exists z\ P(x,z)\ \&\ P(y,z)) \rightarrow \exists u\ Q(x,y,u)$$

4.14 Prove that the formula $\exists x\ P(x) \rightarrow \forall x\ P(x)$ is always true if the domain D contains only one element.

4.15 Find the standard normal form for the following statement.

$$\forall x\ (\neg P(x,0) \rightarrow (\exists y\ (P(y,g(x))\ \&\ \forall z\ (P(z,g(x)) \rightarrow P(y,z))$$

4.16 Referring to Problems 4.4 and 4.5, use resolution to show (by refutation) that you can get from Town-E to Town-C.

4.17 Write a LISP resolution program to answer different queries about a knowledge base where the knowledge base and queries are given in clausal form. For example,

```
(setq KB (A1 to A13))
(query (GETTO Town-E Town-C))
```

4.18 Given the following information for a database:
A1. If x is on top of y, y supports x.
A2. If x is above y and they are touching each other, x is on top of y.
A3. A cup is above a book.
A4. A cup is touching a book.
(a) Translate statements A1 through A4 into clausal form.
(b) Show that the predicate supports (book, cup) is true using resolution.

4.19 Write a PROLOG program using A1 to A4 of Problem 4.18, and show that SUPPORTS (book cup) is true.

5

Dealing with Inconsistencies and Uncertainties

The previous chapter considered methods of reasoning under conditions of certain, complete, unchanging, and consistent facts. It was implicitly assumed that a sufficient amount of reliable knowledge (facts, rules, and the like) was available with which to deduce confident conclusions. While this form of reasoning is important, it suffers from several limitations.

1. It is not possible to describe many envisioned or real-world concepts; that is, it is limited in expressive power.
2. There is no way to express uncertain, imprecise, hypothetical or vague knowledge, only the truth or falsity of such statements.
3. Available inference methods are known to be inefficient.
4. There is no way to produce new knowledge about the world. It is only possible to add what is derivable from the axioms and theorems in the knowledge base.

In other words, strict classical logic formalisms do not provide realistic representations of the world in which we live. On the contrary, intelligent beings are continuously required to make decisions under a veil of uncertainty.

Uncertainty can arise from a variety of sources. For one thing, the information

we have available may be incomplete or highly volatile. Important facts and details which have a bearing on the problems at hand may be missing or may change rapidly. In addition, many of the "facts" available may be imprecise, vague, or fuzzy. Indeed, some of the available information may be contradictory or even unbelievable. However, despite these shortcomings, we humans miraculously deal with uncertainties on a daily basis and usually arrive at reasonable solutions. If it were otherwise, we would not be able to cope with the continually changing situations of our world.

In this and the following chapter, we shall discuss methods with which to accurately represent and deal with different forms of inconsistency, uncertainty, possibility, and beliefs. In other words, we shall be interested in representations and inference methods related to what is known as commonsense reasoning.

5.1 INTRODUCTION

Consider the following real-life situation. Timothy enjoys shopping in the mall only when the stores are not crowded. He has agreed to accompany Sue there on the following Friday evening since this is normally a time when few people shop. Before the given date, several of the larger stores announce a one-time, special sale starting on that Friday evening. Timothy, fearing large crowds, now retracts the offer to accompany Sue, promising to go on some future date. On the Thursday before the sale was to commence, weather forecasts predicted heavy snow. Now, believing the weather would discourage most shoppers, Timothy once again agreed to join Sue. But, unexpectedly, on the given Friday, the forecasts proved to be false; so Timothy once again declined to go.

This anecdote illustrates how one's beliefs can change in a dynamic environment. And, while one's beliefs may not fluctuate as much as Timothy's, in most situations, this form of belief revision is not uncommon. Indeed, it is common enough that we label it as a form of commonsense reasoning, that is, reasoning with uncertain knowledge.

Nonmonotonic Reasoning

The logics we studied in the previous chapter are known as monotonic logics. The conclusions derived using such logics are valid deductions, and they remain so. Adding new axioms increases the amount of knowledge contained in the knowledge base. Therefore, the set of facts and inferences in such systems can only grow larger; they can not be reduced; that is, they increase monotonically. The form of reasoning performed above by Timothy, on the other hand, is nonmonotonic. New facts became known which contradicted and invalidated old knowledge. The old knowledge was retracted causing other dependent knowledge to become invalid, thereby requiring further retractions. The retractions led to a shrinkage or nonmonotonic growth in the knowledge at times.

More formally, let KBL be a formal first order system consisting of a knowledge base and some logic L. Then, if KB1 and KB2 are knowledge bases where

KB1 = KBL
KB2 = KBL ∪ F, for some wff F, then
KB1 ⊆ KB2

In other words, a first order KB system can only grow monotonically with added knowledge.

When building knowledge-based systems, it is not reasonable to expect that all the knowledge needed for a set of tasks could be acquired, validated, and loaded into the system at the outset. More typically, the initial knowledge will be incomplete, contain redundancies, inconsistencies, and other sources of uncertainty. Even if it were possible to assemble complete, valid knowledge initially, it probably would not remain valid forever, not in a continually changing environment.

In an attempt to model real-world, commonsense reasoning, researchers have proposed extensions and alternatives to traditional logics such as PL and FOPL. The extensions accommodate different forms of uncertainty and nonmonotony. In some cases, the proposed methods have been implemented. In other cases they are still topics of research. In this and the following chapter, we will examine some of the more important of these methods.

We begin in the next section with a description of truth maintenance systems (TMS), systems which have been implemented to permit a form of nonmonotonic reasoning by permitting the addition of changing (even contradictory) statements to a knowledge base. This is followed in Section 5.3 by a description of other methods which accommodate nonmonotonic reasoning through default assumptions for incomplete knowledge bases. The assumptions are plausible most of the time, but may have to be retracted if other conflicting facts are learned. Methods to constrain the knowledge that must be considered for a given problem are considered next. These methods also relate to nonmonotonic reasoning. Section 5.4 gives a brief treatment of modal and temporal logics which extend the expressive power of classical logics to permit representations and reasoning about necessary and possible situations, temporal, and other related situations. Section 5.5 concludes the chapter with a brief presentation of a relatively new method for dealing with vague and imprecise information, namely fuzzy logic and language computation.

5.2 TRUTH MAINTENANCE SYSTEMS

Truth maintenance systems (also known as belief revision and revision maintenance systems), are companion components to inference systems. The main job of the TMS is to maintain consistency of the knowledge being used by the problem solver and not to perform any inference functions. As such, it frees the problem solver from any concerns of consistency and allows it to concentrate on the problem solution

aspects. The TMS also gives the inference component the latitude to perform nonmonotonic inferences. When new discoveries are made, this more recent information can displace previous conclusions that are no longer valid. In this way, the set of beliefs available to the problem solver will continue to be current and consistent.

Figure 5.1 illustrates the role played by the TMS as part of the problem solver. The inference engine (IE) solves domain problems based on its current belief set, while the TMS maintains the currently active belief set. The updating process is incremental. After each inference, information is exchanged between the two components. The IE tells the TMS what deductions it has made. The TMS, in turn, asks questions about current beliefs and reasons for failures. It maintains a consistent set of beliefs for the IE to work with even if new knowledge is added and removed.

For example, suppose the knowledge base (KB) contained only the propositions P, $P \rightarrow Q$, and modus ponens. From this, the IE would rightfully conclude Q and add this conclusion to the KB. Later, if it was learned that $\neg P$ was appropriate, it would be added to the KB resulting in a contradiction. Consequently, it would be necessary to remove P to eliminate the inconsistency. But, with P now removed, Q is no longer a justified belief. It too should be removed. This type of belief revision is the job of the TMS.

Actually, the TMS does not discard conclusions like Q as suggested. That could be wasteful, since P may again become valid, which would require that Q and facts justified by Q be rederived. Instead, the TMS maintains dependency records for all such conclusions. These records determine which set of beliefs are current (which are to be used by the IE). Thus, Q would be removed from the current belief set by making appropriate updates to the records and not by erasing Q. Since Q would not be lost, its rederivation would not be necessary if P became valid once again.

The TMS maintains complete records of reasons or justifications for beliefs. Each proposition or statement having at least one valid justification is made a part of the current belief set. Statements lacking acceptable justifications are excluded from this set. When a contradiction is discovered, the statements responsible for the contradiction are identified and an appropriate one is retracted. This in turn may result in other retractions and additions. The procedure used to perform this process is called dependency-directed backtracking. This process is described later.

The TMS maintains records to reflect retractions and additions so that the IE

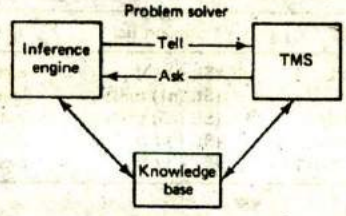


Figure 5.1 Architecture of the problem solver with a TMS.

will always know its current belief set. The records are maintained in the form of a dependency network. The nodes in the network represent KB entries such as premises, conclusions, inference rules, and the like. Attached to the nodes are justifications which represent the inference steps from which the node was derived. Nodes in the belief set must have valid justifications. A premise is a fundamental belief which is assumed to be always true. Premises need no justifications. They form a base from which all other currently active nodes can be explained in terms of valid justifications.

There are two types of justification records maintained for nodes: support lists (SL) and conceptual dependencies (CP). SLs are the most common type. They provide the supporting justifications for nodes. The data structure used for the SL contains two lists of other dependent node names, an in-list and an out-list. It has the form

(SL <in-list> <out-list>)

In order for a node to be active and, hence, labeled as IN the belief set, its SL must have at least one valid node in its in-list, and all nodes named in its out-list, if any, must be marked OUT of the belief set. For example, a current belief set that represents Cybil as a nonflying bird (an ostrich) might have the nodes and justifications listed in Table 5.1.

Each IN-node given in Table 5.1 is part of the current belief set. Nodes n1 and n5 are premises. They have empty support lists since they do not require justifications. Node n2, the belief that Cybil *can* fly is out because n3, a valid node, is in the out-list of n2.

Suppose it is discovered that Cybil is not an ostrich, thereby causing n5 to be retracted (marking its status as OUT). Then n3, which depends on n5, must also be retracted. This, in turn, changes the status of n2 to be a justified node. The resultant belief set is now that the bird Cybil can fly.

To represent a belief network, the symbol conventions shown in Figure 5.2 are sometimes used. The meanings of the nodes shown in the figure are (1) a premise is a true propositon requiring no justification, (2) an assumption is a current belief that could change, (3) a datum is either a currently assumed or IE derived belief, and (4) justifications are the belief (node) supports, consisting of supporting antecedent node links and a consequent node link.

TABLE 5.1 EXAMPLE NODES IN A DEPENDENCY NETWORK

Node	Status	Meaning	Support list	Comments
n1	IN	Cybil is a bird	(SL ( ) ( ))	a premise
n2	OUT	Cybil can fly	(SL (n1) (n3))	unjustified belief
n3	IN	Cybil cannot fly	(SL (n5) (n4))	justified belief
n4	OUT	Cybil has wings	(SL ( ) ( ))	retracted premise
n5	IN	Cybil is an Ostrich	(SL ( ) ( ))	a premise

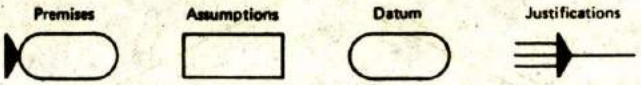


Figure 5.2 Belief network node meanings.

An example of a typical network representation is given in Figure 5.3. Note that nodes T, U, and W are OUT since they lack needed support from P. If the node labeled P is made IN for some reason, the TMS would update the network by propagating the "inness" support provided by node P to make T, U, and W IN.

As noted earlier, when a contradiction is discovered, the TMS locates the source of the contradiction and corrects it by retracting one of the contributing sources. It does this by checking the support lists of the contradictory node and going directly to the source of the contradiction. It goes directly to the source by examining the dependency structure supporting the justification and determining the offending nodes. This is in contrast to the naive backtracking approach which would search a deduction tree sequentially, node-by-node until the contradictory node is reached. Backtracking directly to the node causing the contradiction is known as dependency-directed backtracking (DDB). This is clearly a more efficient search strategy than chronological backtracking. This process is illustrated in Figure 5.4 where it is assumed that A and D are contradictory. By backtracking directly to

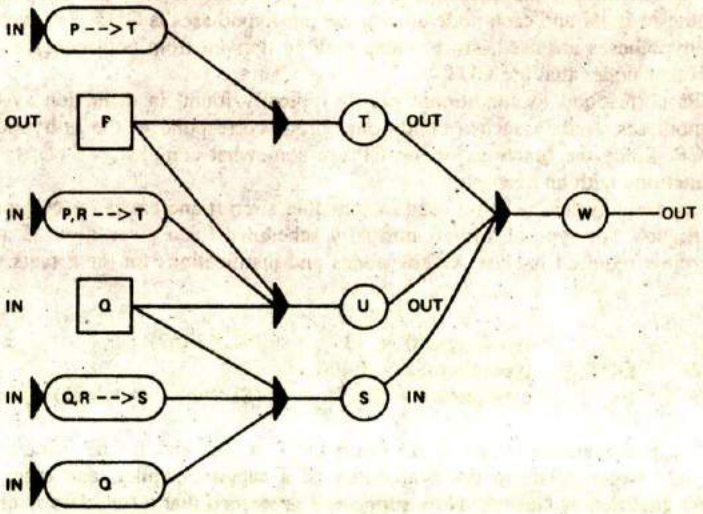


Figure 5.3 Typical fragment of a belief network.

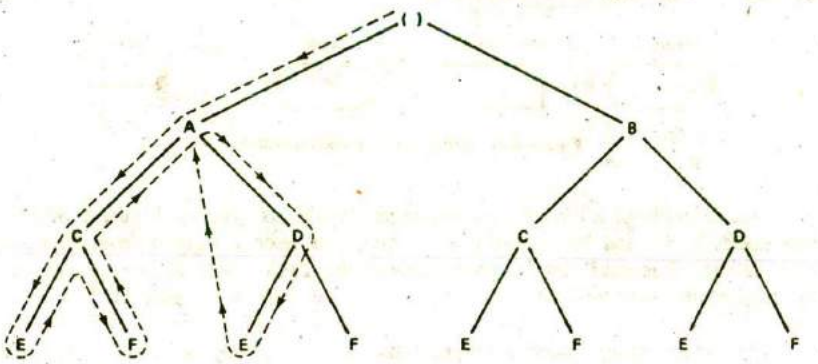


Figure 5.4 Dependency-directed backtracking in a TMS.

the source of a contradiction (the dashed line from E to A), extra search time is avoided.

CP justifications are used less frequently than the SLs. They justify a node as a type of valid hypothetical argument. The internal form of a CP justification is as follows:

(CP <consequent><inhypotheses><outhypotheses>)

A CP is valid if the consequent node is IN whenever each node among the in-hypotheses is IN and each node among the out-hypotheses is OUT. Two separate lists of hypotheses are used, since nodes may be derived from both nodes that are IN and other nodes that are OUT.

CPs correspond to conditional proofs typically found in deduction systems. The hypotheses used in such a conditional proof correspond to the in-hypotheses in the CP. Since the functions of the CP are somewhat complex, we clarify their main functions with an example.

Suppose a system is being used to schedule aircraft and crews for commercial airline flights. The type of aircraft normally scheduled for a given flight is a 737, and the crew required is class A. The nodes and justifications for these facts might be

n1	IN	type(aircraft) = 737	(SL ( ) (n2))
n2	OUT	type(aircraft) = L400	
n3	IN	class(crew) = A	(SL (n8, . . . , n22) ( ))

where the justifications for n1 is n2 (with OUT status) and for n3 is nodes n8, . . . , n22 which relate to the availability of a captain, copilot, and other crew members qualified as class A. Now suppose it is learned that a full class A crew is not available. To complete a schedule for the flight, the system must choose an

alternative aircraft, say an L400. To do this the IE changes the status of n2 to IN. But this results in a contradiction and, hence, creation of the node

```
n4    IN    contradiction    (SL (n1,n3) ( ))
```

The contradiction now initiates the DDB procedure in the TMS to locate the offending assumptions. Since there is only one such node, n1, its retraction is straightforward. For this, the TMS creates a "nogood" node with a CP justification as

```
n5    IN    nogood n1    (CP n4 (n1,n3) ( ))
```

To correct the inconsistency, the TMS next makes n2 an IN node by justifying it with n5 as follows

```
n2    IN    type(aircraft) = L400    (SL (n5) ( ))
```

This in turn causes n1 to become OUT (as an assumption n1 has a nonempty out-list). Also, since n4 was justified by n1, it too must become OUT. This gives the following set of nodes

```
n1    OUT   type(aircraft) = 737     (SL ( ) (n2))
n2    IN    type(aircraft) = L400    (SL (n5) ( ))
n3    IN    class(crew) = A          (SL (n8, . . . , n22) ( ))
n4    OUT   contradiction            (SL (n1,n3) ( ))
n5    IN    nogood n1                (CP n4 (n1,n3) ( ))
```

Note that a CP justification was needed for the "nogood" node to prevent a circular retraction of n2 from occurring. Had an SL been used for n5 with an n4 node in-list justification, n5 would have become OUT after n4, again causing n2 to become OUT.

The procedures for manipulating CPs are quite complicated, and, since they are usually converted into SLs anyway, we only mention their main functions here. For a more detailed account see Doyle (1979).

We have briefly described the JTMS here since it is the simplest and most widely used truth maintenance system. This type of TMS is also known as a nonmonotonic TMS (NMTMS). Several other types have been developed to correct some of the deficiencies of the JTMS and to meet other requirements. They include the logic-based TMS (the LTMS), and the assumption-based TMS (the ATMS), as well as others (de Kleer, 1986a and 1986b).

5.3 DEFAULT REASONING AND THE CLOSED WORLD ASSUMPTION

Another form of uncertainty occurs as a result of incomplete knowledge. One way humans deal with this problem is by making plausible default assumptions; that is, we make assumptions which typically hold but may have to be retracted if new

information is obtained to the contrary. For example, if you have an acquaintance named Pat who is over 20 years old, you would normally assume that this person would drive a car. Later, if you learned Pat had suffered from blackouts until just recently, you would be forced to revise your beliefs.

Default Reasoning

Default reasoning is another form of nonmonotonic reasoning; it eliminates the need to explicitly store all facts regarding a situation. Reiter (1980) develops a theory of default reasoning within the context of traditional logics. A default is expressed as

$$\frac{a(x) : Mb_1(x), \ldots, Mb_k(x)}{c(x)} \tag{5.1}$$

where $a(x)$ is a precondition wff for the conclusion wff $c(x)$, M is a consistency operator and the $b_i(x)$ are conditions, each of which must be separately consistent with the KB for the conclusion $c(x)$ to hold. As an example, suppose we wish to make the statement, "If x is an adult and it is consistent to assume that x can drive, then infer that x can drive." Using the above formula this would be represented as

$$\frac{\text{ADULT}(x) : \text{MDRIVE}(x)}{\text{DRIVE}(x)}$$

Default theories consist of a set of axioms and set of default inference rules with schemata like formula 5.1. The theorems derivable from a default system are those that follow from first-order logic and the assumptions assumed from the default rules.

Suppose a KB contains only the statements

$$\frac{\text{BIRD}(x) : \text{MFLY}(x)}{\text{FLY}(x)}$$

$$\text{BIRD(tweety)}$$

A default proof of FLY(tweety) is possible. But if KB also contains the clauses

$$\text{OSTRICH(tweety)}$$

$$\text{OSTRICH}(x) \rightarrow \neg\text{FLY}(x)$$

FLY(tweety) would be blocked since the default is now inconsistent.

Default rules are especially useful in hierarchial KBs. Because the default rules are transitive, property inheritance becomes possible. For example, in a heirarchy of living things, any animal could inherit the property has-heart from the rule

$$\forall x \ \text{ANIMAL}(x) \rightarrow \text{HAS-HEART}(x)$$

Transitivity can also be a problem in KBs with many default rules. Rule interactions can make representations very complex. Therefore caution is needed in implementing such systems.

Closed World Assumption

Another form of assumption, made with regard to incomplete knowledge, is more global in nature than single defaults. This type of assumption is useful in applications where most of the facts are known, and it is, therefore, reasonable to assume that if a proposition cannot be proven, it is false. This is known as the closed world assumption (CWA) with failure as negation (failure to prove a statement F results in assuming its negation, $\neg F$). This means that in a KB if the ground literal $P(a)$ is not provable, then $\neg P(a)$ is assumed to hold.

A classic example where this type of assumption is reasonable is in an airline KB application where city-to-city flights, not explicitly entered or provable, are assumed not to exist. Thus, ¬CONNECT(boston,van-horn) would be inferred whenever CONNECT(boston,van-horn) could not be derived from the KB. This seems reasonable since we would not want to enter all pairs of cities which do not have intercity flights.

By augmenting a KB with an assumption (a metarule) which states that if the ground atom $P(a)$ cannot be proved, assume its negation $\neg P(a)$, the CWA completes the theory with respect to KB. (Recall that a formal KB system is complete if and only if every ground atom or its negation is in the system.) Augmenting a KB with the negation of all ground atoms of the language which are not derivable, gives us a complete theory. For example, a KB containing only the clauses

$$P(a)$$

$$P(b)$$

$$P(a) \rightarrow Q(a)$$

and modus ponens is not complete, since neither $Q(b)$ nor $\neg Q(b)$ is inferrable from the KB. This KB can be completed, however, by adding either $Q(b)$ or $\neg Q(b)$.

In general, a KB augmented with CWA is not consistent. This is easily seen by considering the KB consisting of the clause

$$P(a) \vee Q(b)$$

Now, since none of the ground literals in this clause are derivable, the augmented KB becomes

$$P(a) \vee Q(b)$$

$$\neg P(a), \neg Q(b)$$

which is inconsistent.

It can be shown that consistency can be maintained for a special type of

CWA. This is a KB consisting of Horn clauses. If a KB is consistent and Horn, then its CWA augmentation is consistent. (Recall that Horn clauses are clauses with at most one positive literal.)

It may appear that the global nature of the negation as failure assumption is a serious drawback of CWA. And indeed there are many applications where it is not appropriate. There is a way around this using completion formulas which we discuss in the next section. Even so, CWA is essentially the formalism under which Prolog operates and Prolog has been shown to be effective in numerous applications.

5.4 PREDICATE COMPLETION AND CIRCUMSCRIPTION

Limiting default assumptions to only portions of a KB can be achieved through the use of completion or circumscription formulas. Unlike CWA, these formulas apply only to specified predicates, and not globally to the whole KB.

Completion Formulas

Completion formulas are axioms which are added to a KB to restrict the applicability of specific predicates. If it is known that only certain objects should satisfy given predicates, formulas which make this knowledge explicit (complete) are added to the KB. This technique also requires the addition of the unique-names assumption (UNA); that is, formulas which state that distinguished named entities in the KB are unique (different).

As an example of predicate completion, suppose we have the following KB:

OWNS(joe,ford)
STUDENT(joe)
OWNS(jill,chevy)
STUDENT(jill)
OWNS(sam,bike)
PROGRAMMER(sam)
STUDENT(mary)

If it is known that Joe is the only person who owns a Ford, this fact can be made explicit with the following completion formula:

$$\forall x \; \text{OWNS}(x,\text{ford}) \rightarrow \text{EQUAL}(x,\text{joe}) \tag{5.2}$$

In addition, we add the inequality formula

$$\neg\text{EQUAL}(a,\text{joe}) \tag{5.3}$$

which has the meaning that this is true for all constants a which are different from Joe.

Likewise, if it is known that Mary also has a Ford, and only Mary and Joe have Fords, the completion and corresponding inequality formulas in this case would be

$$\forall x\ \text{OWNS}(\text{ford},x) \rightarrow \text{EQUAL}(x,\text{joe}) \vee \text{EQUAL}(x,\text{mary})$$
$$\neg\text{EQUAL}(a,\text{joe})$$
$$\neg\text{EQUAL}(a,\text{mary})$$

Once completion formulas have been added to a KB, ordinary first order proof methods can be used to prove statements such as ¬OWNS(jill,ford). For example, to obtain a refutation proof using resolution, we put the completion and inequality formulas 5.2 and 5.3 in clausal form respectively, negate the query ¬OWNS(jill,ford) and add it to the KB. Thus, resolving with the following clauses

1. ¬OWNS(x,ford) V EQUAL(x,joe)
2. ¬EQUAL(a,joe)
3. OWNS(jill,ford)

from 1 and 3 we obtain

4. EQUAL(jill,joe)

and from 2 and 4 (recall that a is any constant not = Joe) we obtain the empty clause [ ], proving the query.

The need for the inequality formulas should be clearer now. They are needed to complete the proof by restricting the objects which satisfy the completed predicates.

Predicate completion performs the same function as CWA but with respect to the completed predicates only. However with predicate completion, it is possible to default both negative as well as positive statements.

Circumscription

Circumscription is another form of default reasoning introduced by John McCarthy (1980). It is similar to predicate completion in that all objects that can be shown to have some property P are in fact the only objects that satisfy P. We can use circumscription, like predicate completion, to constrain the objects which must be considered in any given situation. Suppose we have a university world situation in which there are two known CS students. We wish to state that the known students are the only students, to "circumscribe" the students

$$\text{CSSTUDENT}(a)$$
$$\text{CSSTUDENT}(b)$$

Let x be a tuple, that is $x = (x_1, \ldots, x_n)$, and let ϕ denote a relation of the same arity as P. Also let $\text{KB}(\phi(x))$ denote a KB with every occurrence of P

in KB replaced by ϕ. The usual form of circumscribing P in KB, denoted as CIR(KB:P), is given by

$$CIR(KB:P) = KB \ \& \ [\forall \phi (KB(\phi) \ \& \ (\forall x \phi(x) \rightarrow P(x))] \rightarrow \forall x [P(x) \rightarrow \phi(x)]$$

This amounts to replacing KB with a different KB in which some expression ϕ (in this example $\phi = (x = a \vee x = b)$) replaces each occurrence of P (here CSSTUDENT). The first conjunct inside the bracket identifies the required substitution. The second conjunct, $\forall x \phi(x) \rightarrow P(x)$, states that objects which satisfy ϕ also satisy P, while the conclusion states that ϕ and P are then equivalent.

Making the designated substitution in our example we obtain the circumscriptive inference

From CSSTUDENT(a) & CSSTUDENT(b), infer

$$\forall x (\text{CSSTUDENT}(x) \rightarrow (x = a \vee x = b)$$

Note that ϕ has been quantified in the circumscription schema above, making it a second order formula. This need not be of too much concern since in many cases the formula can be rewritten in equivalent first order form.

An interesting example used by McCarthy in motivating circumscription, related to the task of getting across a river when only a row boat was available. For problems such as this, it should only be necessary to consider those objects named as being important to the task completion and not innumerable other contingencies, like a hole in the boat, lost oars, breaking up on a rock, or building a bridge.

5.5 MODAL AND TEMPORAL LOGICS

Modal logics were invented to extend the expressive power of traditional logics. The original intent was to add the ability to express the necessity and possibility of propositions P in PL and FOPL. Later, other modal logics were introduced to help capture and represent additional subjective mood concepts (supposition, desire) in addition to the standard indicative mood (factual) concept representations given by PL and FOPL.

With modal logics we can also represent possible worlds in addition to the actual world in which we live. Thus, unlike traditional logics, where an interpretation of a wff results in an assignment of true or false (in one world only), an interpretation in modal logics would be given for each possible world. Consequently, the wff may be true in some worlds and false in others.

Modal logics are derived as extensions of PL and FOPL by adding modal operators and axioms to express the meanings and relations for concepts such as consistency, possibility, necessity, obligation, belief, known truths, and temporal situations, like past, present, and future. The operators take predicates as arguments and are denoted by symbols or letters such as L (it is necessary that), M (it is possible that), and so on. For example, MCOLDER-THAN(denver,portland) would be used to represent the statement "it is possible that Denver is colder than Portland."

Modal logics are classified by the type of modality they express. For example, alethic logics are concerned with necessity and possibility, deontic logics with what is obligatory or permissible, epistemic logics with belief and (known) knowledge, and temporal logics with tense modifiers like sometimes, always, what has been, what will be, or what is. In what follows, we shall be primarily concerned with alethic and temporal logics.

It is convenient to refer to an *agent* as the conceptualization of our knowledge-based system (a robot or some other KB system). We may adopt different views regarding the world or environment in which the agent functions. For example, one view may regard an agent as having a set of basic beliefs which consists of all statements that are derivable from the knowlege base. This was essentially the view taken in PL and FOPL. Note, however, that the statements we now call beliefs are not the same as known factual knowledge of the previous chapter. They may, in fact be erroneous.

In another view, we may treat the agent as having a belief set that is determined by possible worlds which are accessible to the agent. In what follows, we will adopt this latter view. But before describing the modal language of our agent, we should explain further the notion of possible worlds.

Possible Worlds

In different knowledge domains it is often productive to consider possible situations or events as alternatives to actual ones. This type of reasoning is especially useful in fields where an alternative course of action can lead to a significant improvement or to a catastrophic outcome. It is a common form of reasoning in areas like law, economics, politics, and military planning to name a few. Indeed, we frequently engage our imaginations to simulate possible situations which predict outcomes based on different scenarios. Our language has even grown to accommodate hypothetical concepts with statements like "if this were possible," or "suppose we have the following situation." On occasion, we may also wish to think of possible worlds as corresponding to different distributed knowledge bases.

Next, we wish to establish a relationship between an agent A and A's possible worlds. For this, we make use of a binary *accessibility* relation R. R is used to define relative possibilities between worlds for agent A. Let $W = \{w_0, w_1, \ldots\}$ denote a set of possible worlds where w_0 refers to the actual world. Let the relation $R(A: w_i, w_j)$ be defined on W such that for any $w_i, w_j \in W$, w_j is accessible from w_i for agent A whenever R is satisfied. This means that a proposition P is true in w_i if and only if P is true in all worlds accessible from w_i.

Since R is a relation, it can be characterized with relational properties such as reflexivity, transitivity, symmetry, and equivalence. Thus for any $w_i, w_j, w_k \in W$, the following properties are defined.

Reflexive. R is reflexive if $(w_i, w_i) \in R$ for each $w_i \in W$. In other words, all worlds are possible with respect to themselves.

Transitive. R is transitive if when $(w_i,w_j) \in R$, and $(w_j,w_k) \in R$ then $(w_i,w_k) \in R$. If w_j is accessible from w_i and w_k is accessible from w_j, then w_k is accessible from w_i.

Symmetric. R is symmetric if when $(w_i,w_j) \in R$ then $(w_j,w_i) \in R$. If w_j is accessible from w_i, then w_i is accessible from w_j.

Equivalence. R is an equivalence relation if R is reflexive, transitive, and symmetric.

These properties can also be described pictorially as illustrated in Figure 5.5.

Modal Operators and Logics

As suggested above, different operators are used to obtain different modalities. We begin with the standard definitions of a traditional logic, say PL, and introduce appropriate operators and, as appropriate, introduce certain axioms. For example, to define one first order modal logic we designate as L_{LM}, the operators L and M noted above would be included. We also add a necessity axiom which states that if the proposition P is a logical axiom then infer LP (that is, if P is universally valid, it is necessary that P is true). We also define LP = $\neg M \neg P$ or it is necessary that P is true as equivalent to it is not possible that P is not true.

Different modal logics can be obtained from L_{LM} by adding different axioms. Some axioms can also be related to the accessibility relations. Thus, for example, we can say that

If R is reflexive, add LP $\rightarrow$ P. In other words, if it is necessary that P is true in w_i, then P is true in w_i.

If R is transitive, add LP $\rightarrow$ LLP. This states that if P is true in all w_j accessible from some w_i, then P is true in all w_k accessible from w_j.

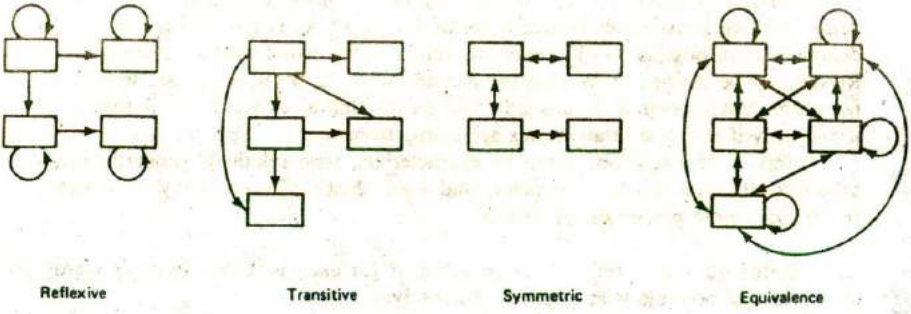


Figure 5.5 Accessibility relation properties.

If R is symmetric, then add P → LMP. This states that if P is true in w_i, then MP is true in all w_j accessible from w_i.

If R is an equivalence relation, add LP → P and MP → LMP.

To obtain a formal system from any of the above, it is necessary to add appropriate inference rules. Many of the inference rules defined in Chapter 4 would apply here as well, (modus ponens, universal substitution, and others). Thus, for a typical modal system based on propositional logic, the following axioms could apply:

A. The basic propositional logic assumptions (Chapter 4) including modus ponens,
B. Addition of the operators L (necessity) and M (possibility), and
C. Some typical modal axioms such as
   1. MP = ¬L¬P (possible P is equivalent to the statement it is not necessary that not P),
   2. L(P → Q) → (LP → LQ) (if it is necessary that P implies Q, then if P is necessary, Q is necessary),
   3. LP → P (if P is necessary then P), and
   4. LP → LLP (if P is necessary, then it is necessary that P is necessary).

Axioms C3 and C4 provide the reflexivity and transitivity access relations, respectively. If other relations are desired, appropriate axioms would be added to those given above; for example, for the symmetric access relation P → LMP would be added.

As an example of a proof in modal logic, suppose some assertions are added to a knowledge base for which the above assumptions and axioms apply:

D. Assertions:
   1. (sam is a man)
   2. M(sam is a child)
   3. L[(sam is a child) → L(sam is a child)]
   4. L[(sam is a man) → ¬(sam is a child)]

A simple proof that ¬(sam is a child) would proceed as follows. From C3 and D4 infer that

(sam is a man) → ¬(sam is a child)

From D1 and E1 using modus ponens conclude

¬(sam is a child)

Temporal Logics

Temporal logics use modal operators in relation to concepts of time, such as past, present, future, sometimes, always, preceeds, succeeds, and so on. An example of two operators which correspond to necessity and possibility are always (A) and

sometimes (S). A propositional temporal logic system using these operators would include the propositional logic assumptions of Chapter 4, the operators A and S, and appropriate axioms. Typical formulas using the A and S operators with the predicate Q would be written as

$AQ \rightarrow SQ$	(if always Q then sometimes Q)
$AQ \rightarrow Q$	(if always Q then Q)
$Q \rightarrow SQ$	(if Q then sometimes Q)
$SQ \rightarrow \neg A \neg Q$	(if sometimes Q then not always not Q)

A combination of propositional and temporal logic inference rules would then be added to obtain a formal system.

In addition to the above operators, the tense operators past (P) and future (F) have also been used. For example, tense formulas for the predicate (tweety files) would be written as

(tweety flies)	present tense.
P(tweety flies)	past tense: Tweety flew.
F(tweety flies)	future tense: Tweety will fly.
FP(tweety flies)	future perfect tense: Tweety will have flown.

It is also possible to define other modalities from the ones given above. For example, to express the concepts it has always been and it will always be the operators H and G may be defined respectively as

$HQ = \neg P \neg Q$	(it has always been the case that Q)
$GQ = \neg F \neg Q$	(it will always be the case that Q)

The four operators P, F, H, and G were used by the logician Lemmon (1965) to define a propositional tense logic he called KT. The language used consisted of propositional logic assumptions, the four tense operators P, F, H, and G, and the following axioms:

$$G(Q \rightarrow R) \rightarrow (GQ \rightarrow GR)$$
$$H(Q \rightarrow R) \rightarrow (HQ \rightarrow HR)$$
$$\neg G \neg HQ \rightarrow Q$$
$$\neg H \neg GQ \rightarrow Q$$

To complete the formal system, inference rules of propositional logic such as modus ponens and the two rules

$$\frac{Q}{HQ} \qquad \frac{Q}{GQ}$$

were added. These rules state that if Q is true, infer that Q has always been the case and Q will always be the case, respectively.

The semantics of a temporal logic is closely related to the frame problem. The frame problem is the problem of managing changes which occur from one situation to another or from frame to frame as in a moving picture sequence.

For example, a KB which describes a robot's world must know which facts change as a result of some action taken by the robot. If the robot throws out the cat, turns off the light, leaves the room, and locks the door, some, but not all facts which describe the room in the new situation must be changed.

Various schemes have been proposed for the management of temporally changing worlds including other modal operators and time and date tokens. This problem has taken on increased importance and a number of solutions have been offered. Still much work remains to find a comprehensive solution.

5.6 FUZZY LOGIC AND NATURAL LANGUAGE COMPUTATIONS

We have already noted several limitations of traditional logics in dealing with uncertain and incomplete knowledge. We have now considered methods which extend the expressive power of the traditional logics and permit different forms of nonmonotonic reasoning. All of the extensions considered thus far have been based on the truth functional features of traditional logics. They admit interpretations which are either true or false only. The use of two valued logics is considered by some practitioners as too limiting. They fail to effectively represent vague or fuzzy concepts.

For example, you no doubt would be willing to agree that the predicate "TALL" is true for Pole, the seven foot basketball player, and false for Smidge the midget. But, what value would you assign for Tom, who is 5 foot 10 inches? What about Bill who is 6 foot 2, or Joe who is 5 foot 5? If you agree 7 foot is tall, then is 6 foot 11 inches also tall? What about 6 foot 10 inches? If we continued this process of incrementally decreasing the height through a sequence of applications of modus ponens, we would eventually conclude that a three foot person is tall. Intuitively, we expect the inferences should have failed at some point, but at what point? In FOPL there is no direct way to represent this type of concept. Furthermore, it is not easy to represent vague statements such as "slightly more beautiful," "not quite as young as . . . ." "not very expensive, but of questionable reliability," "a little bit to the left," and so forth.

Theory of Fuzzy Sets

In standard set theory, an object is either a member of a set or it is not. There is no in between. The natural numbers 3, 11, 19, and 23 are members of the set of prime numbers, but 10, blue, cow, and house are not members. Traditional logics are based on the notions that P(a) is true as long as a is a member of the set belonging to class P and false otherwise. There is no partial containment. This

amounts to the use of a characteristic function f for a set A; where $f_A(x) = 1$ if x is in A; otherwise it is 0. Thus, f is defined on the universe U and for all $x \in U$, $f: U \rightarrow \{0,1\}$.

We may generalize the notion of a set by allowing characteristic functions to assume values other than 0 and 1. For example, we define the notion of a fuzzy set with the characteristic function u which maps from U to a number in the real interval $[0,1]$; that is $u: U \rightarrow [0,1]$. Thus, we define the fuzzy set $\tilde{A}$ as follows (the ~ symbol is omitted but assumed when a fuzzy set appears in a subscript):

Definition. Let U be a set, denumerable or not, and let x be an element of U. A fuzzy subset $\tilde{A}$ of U is a set of ordered pairs $\{(x,u_A(x))\}$, for all x in U, where $u_A(x)$ is a membership characteristic function with values in $[0,1]$, and which indicates the degree or level of membership of x in $\tilde{A}$.

A value of $u_A(x) = 0$ has the same meaning as $f_A(x) = 0$, that x is not a member of $\tilde{A}$, whereas a value of $u_A(x) = 1$ signifies that x is completely contained in $\tilde{A}$. Values of $0 < u_A(x) < 1$ signify that x is a partial member of $\tilde{A}$.

Characteristic functions for fuzzy sets should not be confused with probabilities. A probability is a measure of the degree of uncertainty, likelihood, or belief based on the frequency or proportion of occurrence of an event. Whereas a fuzzy characteristic function relates to vagueness and is a measure of the feasibility or ease of attainment of an event. Fuzzy sets have been related to possibility distributions which have some similarities to probability distributions, but their meanings are entirely different.

Given a definition of a fuzzy set, we now have a means of expressing the notion TALL(x) for an individual x. We might for example, define the fuzzy set $\tilde{A}$ = {tall} and assign values of $u_A(0) = u_A(10) = \ldots = u_A(40) = 0$, $u_A(50) = 0.2$, $u_A(60) = 0.4$, $u_A(70) = 0.6$, $u_A(80) = 0.9$, $u_A(90) = u_A(100) = 1.0$. Now we can assign values for individuals noted above as TALL(Pole) = 1.0, and TALL (Joe) = 0.5. Of course, our assignment of values for $u_A(x)$ was purely subjective. And to be complete, we should also assign values to other fuzzy sets associated with linguistic variables such as very short, short, medium, etc. We will discuss the notions of linguistic variables and related topics later.

Operations on fuzzy sets are somewhat similar to the operations of standard set theory. They are also intuitively acceptable.

$\tilde{A} = \tilde{B}$ if and only if $u_A(x) = u_B(x)$ for all $x \in U$	equality
$\tilde{A} \subseteq \tilde{B}$ if and only if $u_A(x) \leq u_B(x)$ for all $x \in U$	containment
$u_{A \cap B}(x) = \min_x\{u_A(x), u_B(x)\}$	intersection
$u_{A \cup B}(x) = \max_x\{u_A(x), u_B(x)\}$	union
$u_{A'}(x) = 1 - u_A(x)$	complement set

The single quotation mark denotes the complement fuzzy set, A'. Note that the intersection of two fuzzy sets $\tilde{A}$ and $\tilde{B}$ is the largest fuzzy subset that is a subset of

both. Likewise, the union of two fuzzy sets $\tilde{A}$ and $\tilde{B}$ is the smallest fuzzy subset having both $\tilde{A}$ and $\tilde{B}$ as subsets.

With the above definitions, it is possible to derive a number of properties which hold for fuzzy sets much the same as they do for standard sets. For example, we have

$$\tilde{A} \cup (\tilde{B} \cap \tilde{C}) = (\tilde{A} \cup \tilde{B}) \cap (\tilde{A} \cup \tilde{C}) \qquad \text{distributivity}$$

$$\tilde{A} \cap (\tilde{B} \cup \tilde{C}) = (\tilde{A} \cap \tilde{B}) \cup (\tilde{A} \cap \tilde{C})$$

$$(\tilde{A} \cup \tilde{B}) \cup \tilde{C} = \tilde{A} \cup (\tilde{B} \cup \tilde{C}) \qquad \text{associativity}$$

$$(\tilde{A} \cap \tilde{B}) \cap \tilde{C} = \tilde{A} \cap (\tilde{B} \cap \tilde{C})$$

$$\tilde{A} \cap \tilde{B} = \tilde{B} \cap \tilde{A}, \quad \tilde{A} \cup \tilde{B} = \tilde{B} \cup \tilde{A} \qquad \text{commutativity}$$

$$\tilde{A} \cap \tilde{A} = \tilde{A}, \quad \tilde{A} \cup \tilde{A} = \tilde{A} \qquad \text{idempotency}$$

There is also a form of DeMorgan's laws:

$$u_{(A \cap B)'}(x) = u_{A' \cup B'}(x)$$

$$u_{(A \cup B)'}(x) = u_{A' \cap B'}(x)$$

Note however, that

$$\tilde{A} \cap \tilde{A}' \neq \emptyset, \tilde{A} \cup \tilde{A}' \neq U$$

since in general for $u_A(x) = a$, with $0 < a < 1$, we have

$$u_{A \cup A'}(x) = \max[a, 1 - a] \neq 1$$

$$u_{A \cap A'}(x) = \min[a, 1 - a] \neq 0$$

On the other hand the following relations do hold:

$$\tilde{A} \cap \emptyset = \emptyset \qquad \tilde{A} \cup \emptyset = \tilde{A}$$

$$\tilde{A} \cap U = \tilde{A} \qquad \tilde{A} \cup U = U$$

The universe from which a fuzzy set is constructed may also be uncountable. For example, we can define values of u for the fuzzy set $\tilde{A}$ = {young} as

$$u_A(x) = \begin{cases} 1.0 & \text{for } 0 \leq x \leq 20 \\ \left[1 + \left(\dfrac{x - 20}{10}\right)^2\right]^{-1} & \text{for } x > 20 \end{cases}$$

The values of $u_A(x)$ are depicted graphically, in Figure 5.6.

A number of operations that are unique to fuzzy sets have been defined. A few of the more common operations include

Dilation. The dilation of $\tilde{A}$ is defined as

$$\text{DIL}(\tilde{A}) = [u_A(x)]^{1/2} \qquad \text{for all } x \text{ in } U$$

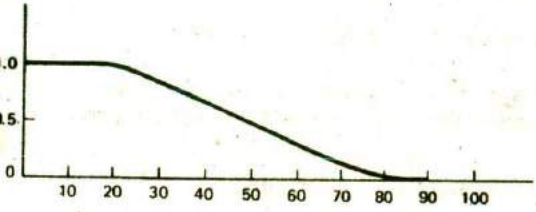


Figure 5.6 Degrees of membership for young age.

Concentration. The concentration of $\tilde{A}$ is defined as

$$\text{CON}(\tilde{A}) = [u_A(x)]^2 \qquad \text{for all } x \text{ in } U$$

Normalization. The normalization of $\tilde{A}$ is defined as

$$\text{NORM}(\tilde{A}) = u_A(x) \,/\, \max_x\{u_A(x)\} \qquad \text{for all } x \text{ in } U$$

These three operations are depicted graphically in Figure 5.7. Note that dilation tends to increase the degree of membership of all partial members x by spreading out the characteristic function curve. The concentration is the opposite of dilation. It tends to decrease the degree of membership of all partial members, and concentrates the characteristic function curve. Normalization provides a means of normalizing all characteristic functions to the same base much the same as vectors can be normalized to unit vectors.

In addition to the above, a number of other operations have been defined including fuzzification. Fuzzification permits one to fuzzify any normal set. These operations will not be required here. Consequently we omit their definitions.

Lotfi a. Zadeh of the University of California, Berkeley, first introduced fuzzy sets in 1965. His objectives were to generalize the notions of a set and propositions to accommodate the type of fuzziness or vagueness we have discussed above. Since its introduction in 1965, much research has been conducted in fuzzy set theory and logic. As a result, extensions now include such concepts as fuzzy arithmetic, possibility distributions, fuzzy statistics, fuzzy random variables, and fuzzy set functions.

Many researchers in AI have been reluctant to accept fuzzy logic as a viable alternative to FOPL. Still, successful systems which employ fuzzy logic have been

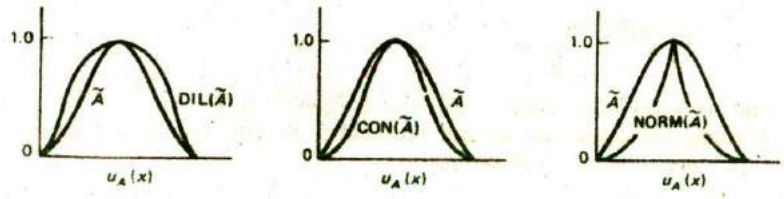


Figure 5.7 Dilation, concentration, and normalization of u.

developed, and a fuzzy VLSI chip has been produced by Bell Telephone Laboratories, Inc., of New Jersey (Togai and Watanabe, 1986).

Reasoning with Fuzzy Logic

The characteristic function for fuzzy sets provides a direct linkage to fuzzy logic. The degree of membership of x in $\tilde{A}$ corresponds to the truth value of the statement x is a member of $\tilde{A}$ where $\tilde{A}$ defines some propositional or predicate class. When $u_A(x) = 1$, the proposition A is completely true, and when $u_A(x) = 0$ it is completely false. Values between 0 and 1 assume corresponding values of truth or falsehood.

In Chapter 4 we found that truth tables were useful in determining the truth value of a statement or wff. In general this is not possible for fuzzy logic since there may be an infinite number of truth values. One could tabulate a limited number of truth values, say those corresponding to the terms false, not very false, not true, true, very true, and so on. More importantly, it would be useful to have an inference rule equivalent to a fuzzy modus ponens.

Generalized modus ponens for fuzzy sets have been proposed by a number of researchers. They differ from the standard modus ponens in that statements which are characterized by fuzzy sets are permitted and the conclusion need not be identical to the implicand in the implication. For example, let $\tilde{A}$, $\tilde{A}1$, $\tilde{B}$, and $\tilde{B}1$ be statements characterized by fuzzy sets. Then one form of the generalized modus ponens reads

Premise: x is $\tilde{A}1$

Implication: If x is $\tilde{A}$ then y is $\tilde{B}$

Conclusion: y is $\tilde{B}1$

An example of this form of modus ponens is given as

Premise: This banana is very yellow

Implication: If a banana is yellow then the banana is ripe

Conclusion: This banana is very ripe

Although different forms of fuzzy inference have been proposed, we present only Zadeh's original compositional rule of inference.

First, recall the definition of a relation. For two sets A and B, the Cartesian product $A \times B$ is the set of all ordered pairs (a,b), for $a \subseteq A$ and $b \subseteq B$. A binary relation on two sets A and B is a subset of $A \times B$. Likewise, we define a binary fuzzy relation $\tilde{R}$ as a subset of the fuzzy Cartesian product $\tilde{A} \times \tilde{B}$, a mapping of $\tilde{A} \rightarrow \tilde{B}$ characterized by the two parameter membership function $u_R(a,b)$. For example, let $\tilde{A} = \tilde{B} = \mathbb{R}$ the set of real numbers, and let $\tilde{R}$: = much larger than. A membership function for this relation might then be defined as

$$u_R(a,b) = \begin{cases} 0 & \text{for } a \leq b \\ (1 + (a-b)^{-2})^{-1} & \text{for } a > b \end{cases}$$

Now let X and Y be two universes and let $\tilde{A}$ and $\tilde{B}$ be fuzzy sets in X and $X \times Y$ respectively. Define fuzzy relations $\tilde{R}_A(x)$, $\tilde{R}_B(x,y)$, and $\tilde{R}_C(y)$ in X, $X \times Y$ and Y, respectively. Then the compositional rule of inference is the solution of the relational equation

$$\tilde{R}_C(y) = \tilde{R}_A(x) \circ \tilde{R}_B(x,y) = \max_x \min\{u_A(x), u_B(x,y)\}$$

where the symbol o signifies the composition of $\tilde{A}$ and $\tilde{B}$. As an example, let

$$X = Y = \{1,2,3,4\}$$

$$\tilde{A} = \{\text{little}\} = \{(1 / 1),(2 / .6),(3 / .2),(4 / 0)$$

$\tilde{R}$ = approximately equal, a fuzzy relation defined by

$\tilde{R}$: x \ y	1	2	3	4
1	1	.5	0	0
2	.5	1	.5	0
3	0	.5	1	.5
4	0	0	.5	1

Then applying the max-min composition rule

$$\begin{aligned}\tilde{R}_C(y) &= \max_x \min\{u_A(x), u_R(x,y)\} \\ &= \max_x \{\min[(1,1),(.6,.5),(.2,0),(0,0)], \\ &\qquad \min[(1,.5),(.6,1),(.2,.5),(0,0)], \\ &\qquad \min[(1,0),(.6,.5),(.2,1),(0,.5)], \\ &\qquad \min[(1,0),(.6,0),(.2,.5),(0,1)]\} \\ &= \max_x\{[1,.5,0,0],[.5,.6,.2,0],[0,.5,.2,0],[0,0,.2,0]\} \\ &= \{[1],[.6],[.5],[.2]\}\end{aligned}$$

Therefore the solution is

$$\tilde{R}_C(y) = \{(1 /1),(2 /.6),(3 /.5),(4/.2)\}.$$

Stated in terms of a fuzzy modus ponens, we might interpret this as the inference

Premise: x is little

Implication: x and y are approximately equal

Conclusion: y is more or less little

The above notions can be generalized to any number of universes by taking the Cartesian product and defining relations on various subsets.

Natural Language Computations

Earlier, we mentioned linguistic variables without really defining them. Linguistic variables provide a link between a natural or artifical language and representations which accommodate quantification such as fuzzy propositions. In this section, we

formally define a linguistic variable and show how they are related to fuzzy logic.

Informally, a linguistic variable is a variable that assumes a value consisting of words or sentences rather than numbers. For example, the variable AGE might assume values of very young, young, not young, or not old. These values, which are themselves linguistic variables, may in turn, each be given meaning through a base universe of elapsed time (years). This is accomplished through appropriately defined characteristic functions.

As an example, let the linguistic variable AGE have possible values {very young, young, not young, middle-aged, not old, old, very old}. To each of these variables, we associate a fuzzy set consisting of ordered tuples $\{(x/u_A(x))\}$ where $x \in U = [0,110]$ the universe (years). The variable AGE, its values, and their corresponding fuzzy sets are illustrated in Figure 5.8.

A formal, more elegant definition of a linguistic variable is one which is based on language theory concepts. For this, we define a linguistic variable as the quintuple

$$(x,T(x),U,G,M)$$

where

x is the name of the variable (AGE in the above example),

$T(x)$ is the terminal set of x (very-young, young, etc.),

U is the universe of discourse (years of life),

G is a set of syntactic rules, the grammer which generates the values of x, and

M is the semantic rule which associates each value of x with its meaning $M(x)$, a fuzzy subset of U.

The grammar G is further defined as the tuple (V_N,V_T,P,S) where V_N is the set of nonterminal symbols, V_T the set of terminal symbols from the alphabet of G, P is

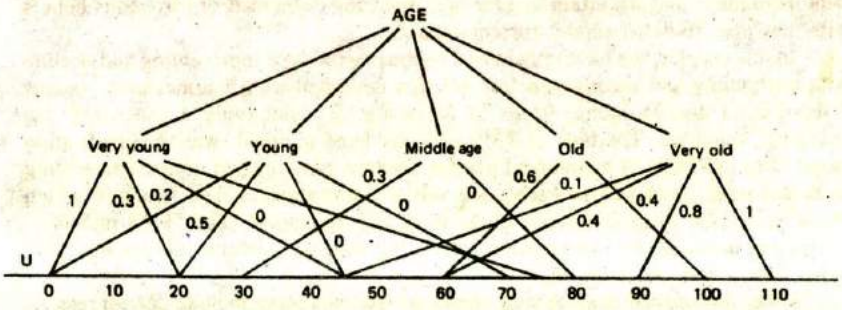


Figure 5.8 The linguistic variable *age*.

a set of rewrite (production) rules, and S is the start symbol. The language $L(G)$ generated by the grammar G is the set of all strings w derived from S consisting of symbols in V_T. Thus, for example, using $V_N = \{A,B,C,D,E,S\}$ and the following rules in P, we generate the terminal string "not young and not very old."

P:	$S \rightarrow A$	$A \rightarrow A$ and B	$C \rightarrow D$	$D \rightarrow$ young
	$S \rightarrow$ S or A	$B \rightarrow C$	$C \rightarrow$ very C	$E \rightarrow$ old
	$A \rightarrow B$	$B \rightarrow$ not C	$C \rightarrow E$	

This string is generated through application of the following rules:

$S \rightarrow A \rightarrow A$ and $B \rightarrow A$ and not $C \rightarrow A$ and not very $C \rightarrow A$ and not very $E \rightarrow A$ and not very old $\rightarrow B$ and not very old $\rightarrow$ not C and not very old $\rightarrow$ not D and not very old $\rightarrow$ not young and not very old

The semantic rule M gives meaning to the values of AGE. For example, we might have $M(\text{old}) = \{(x / u_{\text{old}}(x)) \mid x \in [0,110]\}$ where $u_{\text{old}}(x)$ is defined as

$$u_{\text{old}}(x) = \begin{cases} 0 & \text{for } 0 \leq x \leq 50 \\ \left(1 + \left(\frac{x - 50}{5}\right)^{-2}\right)^{-1} & \text{for } x > 50 \end{cases}$$

In this section we have been able to give only a brief overview of some of the concepts and issues related to the expanding fields based on fuzzy set theory. The interested reader will find numerous papers and texts available in the literature. We offer a few representative references here which are recent and have extensive bibliographies: (Zadeh, 1977, 1981, 1983), (Kandel, 1982), (Gupta et al., (1985), and (Zimmerman, 1985).

5.7 SUMMARY

Commonsense reasoning is nonmonotonic in nature. It requires the ability to reason with incomplete and uncertain knowledge, replacing outmoded or erroneous beliefs with new ones to better model current situations.

In this chapter, we have considered various methods of representing and dealing with uncertainty and inconsistencies. We first described truth maintenance systems which permit nonmonotonic forms of reasoning by maintaining a consistent, but changing, belief set. The type of TMS we considered in detail, was the justification based TMS or JTMS. It maintained a belief network consisting of nodes representing facts and rules. Attached to each node was a justification, a data structure which characterized the In or Out status of the node and its support. The JTMS maintains the current belief set for the problem solver, but does not perform inferencing functions. That is the job of the IE.

Other methods of dealing with nonmonotonic reasoning include default reasoning and CWA. In both cases, assumptions are made regarding knowledge or beliefs which are not directly provable. In CWA, if P can not be proved, then assume $\neg$P

is true. Default reasoning is based on the use of typical assumptions about an object or class of objects which are plausible. The assumptions are regarded as valid unless new contrary information is learned.

Predicate completion and circumscription are methods which restrict the values a predicate or group of predicates may assume. They allow the predicates to take on only those values the KB says they must assume. Both methods are based on the use of completion formulas. Like CWA, they are also a form of nonmonotonic reasoning.

Modal logics extend the expressiveness of classical logics by permitting the notions of possibility, necessity, obligation, belief, and the like. A number of different modal logics have been formalized, and inference rules comparable to propositional and predicate logic are available to permit different forms of nonmonotonic reasoning.

Like modal logics, fuzzy logic was introduced to generalize and extend the expressiveness of traditional logics. Fuzzy logic is based on fuzzy set theory which permits partial set membership. This, together with the ability to use linguistic variables, makes it possible to represent such notions as Sue is not very tall, but she is quite pretty.

EXERCISES

5.1 Give an example of nonmonotonic reasoning you have experienced at some time.

5.2 Draw the TMS belief network for the following knowledge base of facts. The question marks under output status means that the status must be determined for these datum nodes.

INPUTS

Premises	Status	Assumptions	Status
$Q \rightarrow S$	IN	Q	OUT
$Q, R \rightarrow U$	IN	R	IN
$P, R \rightarrow T$	IN		. . .
P	IN		
	. . .		

OUTPUTS

Datum	Status	Conditions
S	?	If Q, $Q \rightarrow S$ then S
U	?	If Q; $Q, R \rightarrow U$, R then U
T	?	If R; $P, R \rightarrow T$; P then T

5.3 Draw a TMS belief network for the aircraft example described in Section 5.2 and show how the network changes with the selection of an alternate choice of aircraft.

5.4 Write schemata for default reasoning using the following statements:

(a) If someone is an adult and it is consistent to assume that adults can vote, infer that that person can vote.

(b) If one is at least 18 years old and it is consistent to assume that one who is physically fit and who passes a test may obtain a pilots license, infer that such a person can obtain a pilots license.

5.5 Show that a Horn clause data base that is consistent is consistent under the CWA assumption. Give an example of a simple Horn clause CWA data base to illustrate consistency.

5.6 For the following database facts, write a completion formula that states that Bill is the only person that lives in Dallas.

LIVESIN(bill,dallas)

LIVESIN(joe,denver)

LIVESIN(sue,phoenix)

OWNS(bill,computer)

STUDENT(sue)

5.7 Determine whether the following modal statements have accessibility relations that are reflexive, transitive, or symmetric:

(a) Bill Brown is alive in the current world.

(b) In the current world and in all worlds in the future of the current world, if Jim Jones is dead in that world, then he will be dead in all worlds in the future of that world.

(c) In the current world or in some world in the future of the current world, John Jones is dead.

5.8 Write modal propositional statements for the following using the operators L and M as described in Section 5.4.

(a) It is necessarily true that the moon is made of green cheese or it is not made of green cheese.

(b) It is possible that if Kennedy were born in Spain Kennedy would speak Spanish.

(c) It is necessarily true that if n is divisible by 4 then n is divisible by 2.

5.9 Show that the dilation of the fuzzy set $\tilde{A} = \text{CON}(\tilde{B})$ is the fuzzy set $\tilde{B}$.

5.10 Give three examples of inferencing with English statements using fuzzy modus ponens (see the example in Section 5.6 under Reasoning with Fuzzy Logic).

5.11 Draw a pictorial definition for the linguistic variable TALL (similar to the variable AGE of Figure 5.8) giving your own subjective values for TALL variables and their values.

5.12 Define a reasonable, real valued fuzzy function for the linguistic variable SHORT (see the function for $\mu_{\text{old}}(x)$).

6

Probabilistic Reasoning

The previous chapter considered methods of representation which extend the expressiveness of classical logic and permit certain types of nonmonotonic reasoning. Representations for vague and imprecise concepts were also introduced with fuzzy set theory and logic. There are other types of uncertainty induced by random phenomena which we have not yet considered. To round out the approaches which are available for commonsense reasoning in AI, we continue in this chapter with theory and methods used to represent probabilistic uncertainties.

6.1 INTRODUCTION

We saw in the previous chapter that a TMS deals with uncertainty by permitting new knowledge to replace old knowledge which is believed to be outdated or erroneous. This is not the same as inferring directly with knowledge that can be given a probability rating based on the amount of uncertainty present. In this chapter, we want to examine methods which use probabilistic representations for all knowledge and which reason by propagating the uncertainties from evidence and assertions to conclusions. As before, the uncertainties can arise from an inability to predict outcomes due to unreliable, vague, incomplete, or inconsistent knowledge.

The probability of an uncertain event A is a measure of the degree of likelihood

of occurrence of that event. The set of all possible events is called the sample space, S. A probability measure is a function $P(\cdot)$ which maps event outcomes E_1, E_2, . . . , from S into real numbers and which satisfies the following axioms of probability:

1. $0 \leq P(A) \leq 1$ for any event $A \subseteq S$.
2. $P(S) = 1$, a certain outcome.
3. For $E_i \cap E_j = \emptyset$, for all $i \neq j$ (the E_i are mutually exclusive), $P(E_1 \cup E_2 \cup E_3 \cup \ldots) = P(E_1) + P(E_2) + P(E_3) + \ldots$

From these three axioms and the rules of set theory, the basic laws of probability can be derived. Of course, the axioms are not sufficient to compute the probability of an outcome. That requires an understanding of the underlying distributions which must be established through one of the following approaches:

1. use of a theoretical argument which accurately characterizes the processes,
2. using one's familiarity and understanding of the basic processes to assign subjective probabilities, or
3. collecting experimental data from which statistical estimates of the underlying distributions can be made.

Since much of the knowledge we deal with is uncertain in nature, a number of our beliefs must be tenuous. Our conclusions are often based on available evidence and past experience, which is often far from complete. The conclusions are, therefore, no more than educated guesses. In a great many situations it is possible to obtain only partial knowledge concerning the possible outcome of some event. But, given that knowledge, one's ability to predict the outcome is certainly better than with no knowledge at all. We manage quite well in drawing plausible conclusions from incomplete knowledge and past experiences.

Probabilistic reasoning is sometimes used when outcomes are unpredictable. For example, when a physician examines a patient, the patient's history, symptoms, and test results provide some, but not conclusive, evidence of possible ailments. This knowledge, together with the physician's experience with previous patients, improves the likelihood of predicting the unknown (disease) event, but there is still much uncertainty in most diagnoses. Likewise, weather forecasters "guess" at tomorrow's weather based on available evidence such as temperature, humidity, barometric pressure, and cloud coverage observations. The physical relationships which govern these phenomena are not fully understood; so predictability is far from certain. Even a business manager must make decisions based on uncertain predictions when the market for a new product is considered. Many interacting factors influence the market, including the target consumer's lifestyle, population growth, potential consumer income, the general economic climate, and many other dependent factors.

In all of the above cases, the level of confidence placed in the hypothesized conclusions is dependent on the availability of reliable knowledge and the experience of the human prognosticator. Our objective in this chapter is to describe some approaches taken in AI systems to deal with reasoning under similar types of uncertain conditions.

6.2 BAYESIAN PROBABILISTIC INFERENCE

The form of probabilistic reasoning described in this section is based on the Bayesian method introduced by the clergyman Thomas Bayes in the eighteenth century. This form of reasoning depends on the use of conditional probabilities of specified events when it is known that other events have occurred. For two events H and E with the probability $P(E) > 0$, the conditional probability of event H, given that event E has occurred, is defined as

$$P(H|E) = P(H \ \& \ E) / P(E) \tag{6.1}$$

This expression can be given a frequency interpretation by considering a random experiment which is repeated a large number of times, n. The number of occurrences of the event E, say No. (E), and of the joint event H and E, No. $(H \ \& \ E)$, are recorded and their relative frequencies rf computed as

$$rf(H \ \& \ E) = \frac{\text{No. } (A \ \& \ E)}{n} \qquad rf(E) = \frac{\text{No. } (E)}{n} \tag{6.2}$$

When n is large, the two expressions (6.2) approach the corresponding probabilities respectively, and the ratio

$$rf(H \ \& \ E) / rf(E) \approx P(H \ \& \ E) / P(E)$$

then represents the proportion of times event H occurs relative to the occurrence of E, that is, the approximate conditional occurrence of H with E.

The conditional probability of event E given that event H occurred can likewise be written as

$$P(E|H) = P(H \ \& \ E) / P(H) \tag{6.3}$$

Solving 6.3 for $P(H \ \& \ E)$ and substituting this in equation 6.1 we obtain one form of Bayes' Rule

$$P(H|E) = P(E|H)P(H) / P(E) \tag{6.4}$$

This equation expresses the notion that the probability of event H occurring when it is known that event E occurred is the same as the probability that E occurs when it is known that H occurred, multiplied by the ratio of the probabilities of the two events H and E occurring. As an example of the use of equation 6.4, consider the problem of determining the probability that a patient has a certain disease $D1$, given that a symptom E was observed. We wish to find $P(D1|E)$.

Suppose now it is known from previous experience that the prior (unconditional) probabilities $P(D1)$ and $P(E)$ for randomly chosen patients are $P(D1) = 0.05$, and $P(E) = 0.15$, respectively. Also, we assume that the conditional probability of the observed symptom given that a patient has disease $D1$ is known from experience to be $P(E|D1) = 0.95$. Then, we easily determine the value of $P(D1|E)$ as

$$P(D1|E) = P(E|D1)P(D1)/P(E) = (0.95 \times 0.05) / 0.15$$
$$= 0.32$$

It may be the case that the probability $P(E)$ is difficult to obtain. If that is the case, a different form of Bayes' Rule may be used. To see this, we write equation 6.4 with $\neg H$ substituted in place of H to obtain

$$P(\neg H|E) = \frac{P(E|\neg H)P(\neg H)}{P(E)}$$

Next, we divide equation 6.4 by this result to eliminate $P(E)$ and get

$$\frac{P(H|E)}{P(\neg H|E)} = \frac{P(E|H)P(H)}{P(E|\neg H)P(\neg H)} \tag{6.5}$$

Note that equation 6.5 has two terms that are ratios of a probability of an event to the probability of its negation, $P(H|E) / P(\neg H|E)$ and $P(H) / P(\neg H)$. The ratio of the probability of an event E divided by the probability of its negation is called the *odds* of the event and are denoted as $O(E)$. The remaining ratio $P(E|H) / P(E|\neg H)$ in equation 6.5 is known as the likelihood ratio of E with respect to H. We denote this quantity by $L(E|H)$. Using these two new terms, the odds-likelihood form of Bayes' Rule for equation 6.5 may be written as

$$O(H|E) = L(E|H) \cdot O(H)$$

This form of Bayes' Rule suggests how to compute the posterior odds $O(H|E)$ from the prior odds on H, $O(H)$. That value is proportional to the likelihood $L(E|H)$. When $L(E|H)$ is equal to one, the knowledge that E is true has no effect on the odds of H. Values of $L(E|H)$ less than or greater than one decrease or increase the odds correspondingly. When $L(E|H)$ cannot be computed, estimates may still be made by an expert having some knowledge of H and E. Estimating the ratio rather than the individual probabilities appears to be easier for individuals in such cases. This is sometimes done when developing expert systems where more reliable probabilities are not available.

In the example cited above, $D1$ is either true or false, and $P(D1|E)$ is the interpretation which assigns a measure of confidence that $D1$ is true when it is known that E is true. There is a similarity between E, $P(D1|E)$ and modus ponens discussed in Chapter 4. For example, when E is known to be true and $D1$ and E are known to be related, one concludes the truth of $D1$ with a confidence level $P(D1|E)$.

One might wonder if it would not be simpler to assign probabilities to as

many ground atoms $E_1, E_2, \ldots, E_k$ as possible, and compute inferred probabilities (probabilities of $E_j \rightarrow H$ and H) directly from these. The answer is that in general this is not possible. To compute an inferred probability requires a knowledge of the joint distributions of the ground predicates participating in the inference. From the joint distributions, the required marginal distributions may then be computed. The distributions and computations required for that approach are, in general, much more complex than the computations involving the use of Bayes' Rule.

Consider now two events A and $\bar{A}$ which are mutually exclusive ($A \cap \bar{A} = \varnothing$) and exhaustive ($A \cup \bar{A} = S$). The probability of an arbitrary event B can always be expressed as

$$P(B) = P(B \,\&\, A) + P(B \,\&\, \bar{A}) = P(B|A)P(A) + P(B|\bar{A})P(\bar{A})$$

Using this result, equation 6.4 can be written as

$$P(H|E) = P(E|H)P(H) \,/\, [P(E|H)P(H) + P(E|\bar{H})P(\bar{H})] \tag{6.6}$$

Equation 6.6 can be generalized for an arbitrary number of hypotheses H_i, $i = 1, \ldots, k$. Thus, suppose the H_i partition the universe; that is, the H_i are mutually exclusive and exhaustive. Then for any evidence E, we have

$$P(E) = \sum_{i=1}^{k} P(E \,\&\, H_i) = \sum_{i=1}^{k} P(E|H_i)P(H_i)$$

and hence,

$$P(H_i|E) = \frac{P(E|H_i)P(H_i)}{\sum_{j=1}^{k} P(E|H_j)\,P(H_j)} \tag{6.7}$$

Finally, to be more realistic and to accommodate multiple sources of evidence $E_1, E_2, \ldots, E_m$, we generalize equation 6.7 further to obtain

$$P(H_i|E_1,E_2,\ldots,E_m) = \frac{P(E_1,E_2,\ldots,E_m|H_i)P(H_i)}{\sum_{j=1}^{k} P(E_1,E_2,\ldots,E_m|H_j)\,P(H_j)} \tag{6.8}$$

If there are several plausible hypotheses and a number of evidence sources, equation 6.8 can be fairly complex to compute. This is one of the serious drawbacks of the Bayesian approach. A large number of probabilities must be known in advance in order to apply an equation such as 6.8. If there were k hypotheses, H_i, and m sources of evidence, E_j, then $k + m$ prior probabilities must be known in addition to the k likelihood probabilities. The real question then is where does one obtain such a large number of reliable probabilities?

To simplify equation 6.8, it is sometimes assumed that the E_j are statistically independent. In that case, the numerator and denominator probability terms $P(E_1, E_2, \ldots, E_m|H_j)$ factor into

$$P(E_1|H_j)P(E_2|H_j) \ldots P(E_m|H_j)$$

resulting in a somewhat simpler form. But, even though the computations are straightforward, the number of probabilities required in a moderately large system can still be prohibitive, and one may be forced to simply use subjective probabilities when more reliable values are not available. Furthermore, the E_j are almost never completely independent. Consequently, this assumption may introduce intolerable errors.

The formulas presented above suggest how probabilistic evidence would be combined to produce a likelihood estimate for any given hypothesis. When a number of individual hypotheses are possible and several sources of evidence are available, it may be necessary to compute two or more alternative probabilities and select among them. This may mean that none, one, or possibly more than one of the hypotheses could be chosen. Normally, the one having the largest probability of occurrence would be selected, provided no other values were close. Before accepting such a choice, however, it may be desirable to require that the value exceed some threshold to avoid selecting weakly supported hypotheses. In Section 6.5 we describe a typical system which combines similar values and chooses only those conclusions that exceed a threshold of 0.2.

Bayesian Networks

Network representations of knowledge have been used to graphically exhibit the interdependencies which exist between related pieces of knowledge. Much work has been done in this area to develop a formal syntax and semantics for such representations. We discuss related topics in some detail in Chapter 7 when we consider associative networks and conceptual graphs. Here, however, we are more interested in network representations which depict the degrees of belief of propositions and the causal dependencies that exist between them. Inferencing in a network amounts to propogating the probabilities of given and related information through the network to one or more conclusion nodes.

Network representations for uncertain dependencies are further motivated by observations made earlier. If we wish to represent uncertain knowledge related to a set of propositional variables $x_1, \ldots, x_n$ by their joint distribution $P(x_1, \ldots, x_n)$, it will require some 2^n entries to store the distribution explicitly. Furthermore, a determination of any of the marginal probabilities x_i requires summing $P(x_1, \ldots, x_n)$ over the remaining $n - 1$ variables. Clearly, the time and storage requirements for such computations quickly become impractical. Inferring with such large numbers of probabilities does not appear to model the human process either. On the contrary, humans tend to single out only a few propositions which are known to be causally linked when reasoning with uncertain beliefs. This metaphor leads quite naturally to a form of network representation.

One useful way to portray the problem domain is with a network of nodes which represent propositional variables x_i, connected by arcs which represent causal

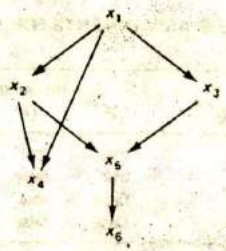


Figure 6.1 Example of Bayesian belief network.

influences or dependencies among the nodes. The strengths of the influences are quantified by conditional probabilities of each variable. For example, to represent causal relationships between the propositional variables $x_1, \ldots, x_6$ as illustrated in Figure 6.1, one can write the joint probability $P(x_1, \ldots, x_6)$ by inspection as a product of (chain) conditional probabilities

$$P(x_1, \ldots, x_6) = P(x_6|x_5)P(x_5|x_2,x_3)P(x_4|x_1,x_2)P(x_3|x_1)P(x_2|x_1)P(x_1)$$

Once such a network is constructed, an inference engine can use it to maintain and propagate beliefs. When new information is received, the effects can be propagated throughout the network until equilibrium probabilities are reached. Pearl (1986, 1987) has proposed simplified methods for updating networks (trees and, more generally, graphs) of this type by fusing and propagating the effects of new evidence and beliefs such that equilibrium is reached in time proportional to the longest path through the network. At equilibrium, all propositions will have consistent probability assignments. Methods for graphs are more difficult. They require the use of dummy variables to transform them to equivalent tree structures which are then easier to work with.

To use the type of probabilistic inference we have been considering, it is first necessary to assign probabilities to all basic facts in the knowledge base. This requires the definition of an appropriate sample space and the assignment of a priori and conditional probabilities. In addition, some method must be chosen to compute the combined probabilities when pooling evidence in a sequence of inference steps (such as Pearl's method). Finally, when the outcome of an inference chain results in one or more proposed conclusions, the alternatives must be compared, and one or more selected on the basis of its likelihood.

6.3 POSSIBLE WORLD REPRESENTATIONS

In Section 5.4 the notion of possible worlds was introduced as a formalism through which an agent could view a set of propositions as being true in some worlds and false in others. We use the possible world concepts in this section to describe a method proposed by Nilsson (1986) which generalizes first order logic in the modelling of uncertain beliefs. The method assigns truth values ranging from 0 to 1 to possible

TABLE 6.1 TRUTH VALUE ASSIGNMENTS FOR THE SET {P, $P \rightarrow Q$, Q}

Consistent			Inconsistent		
P	Q	$P \rightarrow Q$	P	Q	$P \rightarrow Q$
true	true	true	true	true	false
true	false	false	true	false	true
false	true	true	false	true	false
false	false	true	false	false	false

worlds. Each set of possible worlds corresponds to a different interpretation of sentences contained in a knowledge base denoted as KB.

Consider first the simple case where a KB contains only the single sentence S. S may be either true or false. We envision S as being true in one set of possible worlds W_1, and false in another set W_2. The actual world, the one we are in, must be in one of the two sets, but we are uncertain which one. Our uncertainty is expressed by assigning a probability P to W_1 and $1 - P$ to W_2. We can say then that the probability of S being true is P.

When our KB contains L sentences, $S_1, \ldots, S_L$, more sets of possible worlds are required to represent all consistent truth value assignments. There are 2^L possible truth assignments for L sentences. Some of the assignments may be inconsistent (impossible), however. For example, the set of sentences $\{P, P \rightarrow Q, Q\}$ has 2^3 possible truth assignments, only four of which are consistent as depicted in Table 6.1.

If K of the truth assignments are consistent, K sets of possible worlds $W_1, W_2, \ldots, W_K$ are used to model the corresponding interpretations. A probability distribution is then defined over the possible worlds, where P_i is the probability that W_i is the actual world, and $\Sigma_i P_i = 1$. The probability or belief in any sentence S_j can then be determined by summing the P_i over all W_i in which S_j is true.

These notions can be represented using vector notation. We use the matrix $\mathbf{V}$ with L columns and K rows to represent the truth values of the L sentences in each of the K sets W_i. The i^{th} column of $\mathbf{V}$ contains a one in row j if sentence S_j is true in W_i and a zero if it is false. Also let the K-dimensional column vector $\mathbf{p}$ with components p_i, $i = 1, \ldots, K$, represent the possible world probabilities. With this notation, the product of $\mathbf{V}$ and $\mathbf{p}$ is a vector $\mathbf{q}$ which contains the probabilities q_j of sentence S_j, for $j = 1, \ldots, L$, that is

$$\mathbf{q} = \mathbf{V}p \tag{6.9}$$

The j^{th} component q_j of $\mathbf{q}$ is the sum of probabilities of the sets of possible worlds in which S_j is true, or the probabilistic truth value of S_j. As an example of this notation, the matrix equation for the consistent truth value assignments given in Table 6.1 is

$$\mathbf{q} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

where p_1, p_2, p_3, and p_4 are the probabilities for the corresponding W_i. Thus, the sentence probabilities are computed as

$$q_1 = p(S_1) = p_1 + p_2$$
$$q_2 = p(S_2) = p_1 + p_3 + p_4$$
$$q_3 = p(S_3) = p_1 + p_3$$

Given a KB of sentences with known probabilities (obtained from an expert or other source), we wish to determine the probability of any new sentence S deduced from KB. Alternatively, we may wish to recompute some sentence probabilities in KB if new information has been gained which changes one or more of the original sentences in KB. To compute the probability of S requires that consistent truth values first be determined for S for all sets of possible worlds. A new augmented matrix V can then be formed by adding a bottom row of ones and zeros to the original V where the ones and zeros correspond to the truth assignments.

No methods have been developed for the computation of exact solutions for the KB sentence probabilities, although methods for determining approximations were presented for both small and large matrices V. We do not consider those methods here. They may be found in Nilsson (1986). They are based on the use of the probability constraints

$$0 \leq p_i \leq 1, \text{ and } \Sigma_i p_i = 1,$$

and the fact that consistent probability assignments are bounded by the hyperplanes of a certain convex hull. Suggestions have also been made for the partitioning of larger matrices into smaller ones to simplify the computations.

6.4 DEMPSTER-SHAFER THEORY

As noted in the previous section, the assumption of conditional independence is probably not warranted in many problems. There are other serious drawbacks in using Bayesian theory as a model of uncertain reasoning as well. To begin with, the probabilities are described as a single numeric point value. This can be a distortion of the precision that is actually available for supporting evidence. It amounts to an overstatement of the evidence giving support to many of our beliefs. When we assert with probability 0.7 that the dollar will fall against the Japanese Yen over the next six months, what we really mean is we have a fairly strong conviction there is a chance of about 0.6 to 0.8 say, that it will fall.

Another problem with traditional theory is that there is no way to differentiate

between ignorance and uncertainty. These are distinctly different concepts and should be treated as such. For example, suppose we are informed that one of three terrorist groups, A, B, or C has planted a bomb in a certain government building. We may have some evidence to believe that group C is the guilty one and be willing to assign a measure of this belief equal to $P(C) = 0.8$. On the other hand, without more knowledge of the other two groups, we would not want to say that the probability is 0.1 that each one of them is guilty. Yet, traditional theory would have us distribute an equal amount of the remaining probability to each of the other groups. In fact, we may have no knowledge to justify either the amount of uncertainty nor the equal division of it.

Finally, with classical probability theory, we are forced to regard belief and disbelief as functional opposites. That is, if some proposition A is assigned the probability $P(A) = 0.3$, then we must assign $\neg A$ the probability $P(\neg A) = 0.7$ since we must have $P(A) + P(\neg A) = 1$. This forces us to make an assignment that may be conflicting since it is possible to both believe and disbelieve some propositions by the same amount, making this requirement awkward.

In an attempt to remedy the above problems, a generalized theory has been proposed by Arthur Dempster (1968) and extended by his student Glenn Shafer (1976). It has come to be known as the Dempster-Shafer theory of evidence. The theory is based on the notion that separate probability masses may be assigned to all subsets of a universe of discourse rather than just to indivisible single members as required in traditional probability theory. As such, it permits the inequality $P(A) + P(\neg A) \leq 1$.

In the Dempster-Shafer theory, we assume a universe of discourse U and a set corresponding to n propositions, exactly one of which is true. The propositions are assumed to be exhaustive and mutually exclusive. Let 2^U denote all subsets of U including the empty set and U itself (there are 2^n such subsets). Let the set function m (sometimes called a basic probability assignment) defined on 2^U, be a mapping to $[0,1]$,

$$m: 2^U \rightarrow [0,1], \text{ be such that for all subsets } A \subseteq U$$
$$m(\varnothing) = 0$$
$$\sum_{A \subseteq U} m(A) = 1$$

The function m defines a probability distribution on 2^U (not just on the singletons of U as in classical theory). It represents the measure of belief committed exactly to A. In other words, it is possible to assign belief to each subset A of U without assigning any to anything smaller.

A belief function, Bel, corresponding to a specific m for the set A, is defined as the sum of beliefs committed to every subset of A by m. That is, Bel(A) is a measure of the total support or belief committed to the set A and sets a minimum value for its likelihood. It is defined in terms of all belief assigned to A as well as to all proper subsets of A. Thus,

$$\text{Bel}(A) = \sum_{B \subseteq A} m(B)$$

For example, if U contains the mutually exclusive subsets A, B, C, and D then

$$\text{Bel}(\{A,C,D\}) = m(\{A,C,D\}) + m(\{A,C\}) + m(\{A,D\}) + m(\{C,D\}) + m(\{A\}) + m(\{C\}) + m(\{D\})$$

In Dempster-Shafer theory, a belief interval can also be defined for a subset A. It is represented as the subinterval $[\text{Bel}(A), Pl(A)]$ of $[0,1]$. $\text{Bel}(A)$ is also called the *support* of A, and $Pl(A) = 1 - \text{Bel}(\bar{A})$, the *plausibility* of A.

We define $\text{Bel}(\varnothing) = 0$ to signify that no belief should be assigned to the empty set and $\text{Bel}(U) = 1$ to show that the truth is contained within U. The subsets A of U are called the *focal elements* of the support function Bel when $m(A) > 0$.

Since $\text{Bel}(A)$ only partially describes the beliefs about proposition A, it is useful to also have a measure of the extent one believes in $\bar{A}$, that is, the *doubts* regarding A. For this, we define the doubt of A as $D(A) = \text{Bel}(\bar{A})$. From this definition it will be seen that the upper bound of the belief interval noted above, $Pl(A)$, can be expressed as $Pl(A) = 1 - D(A) = 1 - \text{Bel}(\bar{A})$. $Pl(A)$ represents an upper belief limit on the proposition A. The belief interval, $[\text{Bel}(A), Pl(A)]$, is also sometimes referred to as the *confidence* in A, while the quantity $Pl(A) - \text{Bel}(A)$ is referred to as the *uncertainty* in A. It can be shown that (Prade 1983)

$$Pl(\varnothing) = 0,\ Pl(U) = 1$$

For all A,

$$Pl(A) \geq \text{Bel}(A),$$
$$\text{Bel}(A) + \text{Bel}(\bar{A}) \leq 1,$$
$$Pl(A) + Pl(\bar{A}) \geq 1, \text{ and}$$

For $A \subseteq B$,

$$\text{Bel}(A) \leq \text{Bel}(B),\ Pl(A) \leq Pl(B)$$

In interpreting the above definitions, it should be noted that a portion of belief may be committed to a set of propositions, but need not be, and if committed, it is not necessary to commit any belief to its negation. However, a belief committed to a proposition is committed to any other proposition it implies.

A few specific interval belief values will help to clarify the intended semantics. For example,

[0,1]	represents no belief in support of the proposition
[0,0]	represents the belief the proposition is false
[1,1]	represents the belief the proposition is true
[.3,1]	represents partial belief in the proposition
[0,.8]	represents partial disbelief in the proposition
[.2,.7]	represents belief from evidence both for and against the proposition

When evidence is available from two or more independent knowledge sources Bel_1 and Bel_2, one would like to pool the evidence to reduce the uncertainty. For this, Dempster has provided such a combining function denoted as $Bel_1 \circ Bel_2$.

Given two basic probability assignment functions, m_1 and m_2 corresponding to the belief functions Bel_1 and Bel_2, let $A_1, \ldots, A_k$ be the focal elements for Bel_1 and $B_1, \ldots, B_p$ be the focal elements for Bel_2. Then $m_1(A_i)$ and $m_2(B_j)$ each assign probability masses on the unit interval. They can be orthogonally combined as depicted with the square illustrated in Figure 6.2 (Garvey et al., 1981).

The unit square in Figure 6.2 represents the total probability mass assigned by both m_1 and m_2 for all of their common subsets. A particular subrectangle within the square, shown as the intersection of the sets A_i and B_j, has committed to it the measure $m_1(A_i)m_2(B_j)$. Likewise, any subset C of U may have one, or more than one, of these rectangles committed to it. Therefore, the total probability mass committed to C will be

$$\sum_{A_i \cap B_j = C} m_1(A_i)m_2(B_j) \tag{6.9}$$

where the summation is over all i and j.

The sum in equation 6.9 must be normalized to account for the fact that some intersections $A_i \cap B_j = \emptyset$ will have positive probability which must be discarded. The final form of Dempster's rule of combination is then given by

$$m_1 \circ m_2 = \frac{\sum_{A_i \cap B_j} m_1(A_i)m_2(B_j)}{\sum_{A_i \cap B_j \neq 0} m_1(A_i)m_2(B_j)} \tag{6.10}$$

where the summations are taken over all i and j.

As an example of the above concepts, recall once again the problem of identifying the terrorist group or groups responsible for a certain attack in some country. Suppose any of four known terrorist organizations A, B, C, and D could have been responsible for the attack. The possible subsets of U in this case form a lattice of sixteen subsets (Figure 6.3).

Assume one piece of evidence supports the belief that groups A and C were

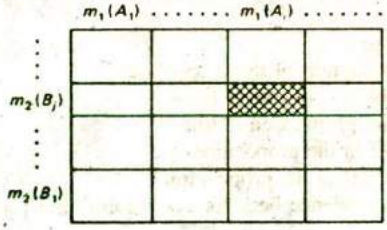


Figure 6.2 Composition of probability mass from sources Bel_1 and Bel_2.

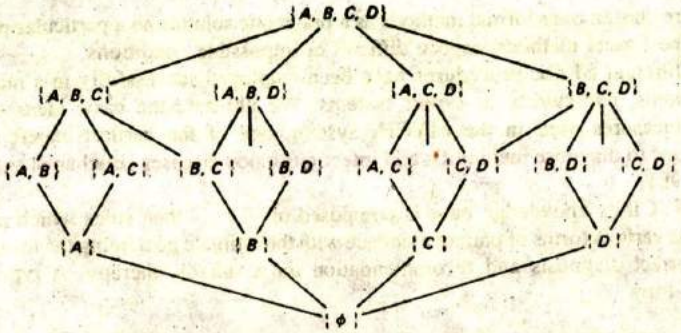


Figure 6.3 Lattice of subsets of the universe U.

responsible to a degree of $m_1(\{A,C\}) = 0.6$, and another source of evidence disproves the belief that C was involved (and therefore supports the belief that the three organizations, A, B, and D were responsible; that is $m_2(\{A,B,D\}) = 0.7$. To obtain the pooled evidence, we compute the following quantities (summarized in Table 6.2).

$$m_1 \circ m_2(\{A\}) = (0.6)*(0.7) = 0.42$$
$$m_1 \circ m_2(\{A,C\}) = (0.6)*(0.3) = 0.18$$
$$m_1 \circ m_2(\{A,B,D\}) = (0.4)*(0.7) = 0.28$$
$$m_1 \circ m_2(\{U\}) = (0.4)*(0.3) = 0.12$$
$$m_1 \circ m_2 = 0 \text{ for all other subsets of U}$$
$$\text{Bel}_1(\{A,C\}) = m(\{A,C\}) + m(\{A\}) + m(\{C\})$$

TABLE 6.2 TABLEAU OF COMBINED VALUES OF BELIEF FOR m_1 AND m_2

		m_2	
		$\{A,B,D\}$ (0.7)	U (0.3)
m_1	$\{A,C\}$ (0.6)	$\{A\}$ (0.42)	$\{A,C\}$ (0.18)
	U (0.4)	$\{A,B,D\}$ (0.28)	U (0.12)

6.5 AD HOC METHODS

The so-called ad hoc methods of dealing with uncertainty are methods which have no formal theoretical basis (although they are usually patterned after probabilistic concepts). These methods typically have an intuitive, if not a theoretical, appeal.

They are chosen over formal methods as a pragmatic solution to a particular problem, when the formal methods impose difficult or impossible conditions.

Different ad hoc procedures have been employed successfully in a number of AI systems, particularly in expert systems. We illustrate the basic ideas with the belief measures used in the MYCIN system, one of the earliest expert systems developed to diagnose meningitis and infectious blood diseases (Buchanan and Shortliffe, 1984).

MYCIN's knowledge base is composed of if . . . then rules which are used to assess various forms of patient evidence with the ultimate goal being the formulation of a correct diagnosis and recommendation for a suitable therapy. A typical rule has the form

IF: The stain of the organism is gram positive, and
The morphology of the organism is coccus, and
The growth conformation of the organism is chains
THEN: There is suggestive evidence (0.7) that the identity of the organism is streptococcus

This is a rule that would be used by the inference mechanism to help identify the offending organism. The three conditions given in the IF part of the rule refer to attributes that help to characterize and identify organisms (the stain, morphology, and growth conformation). When such an identification is relatively certain, an appropriate therapy may then be recommended.

The numeric value (0.7) given in the THEN part of the above rule corresponds to an expert's estimate of degree of belief one can place in the rule conclusion when the three conditions in the IF part have been satisfied. Thus, the belief associated with the rule may be thought of as a (subjective) conditional probability $P(H|E_1,E_2,E_3) = 0.7$, where H is the hypothesis that the organism is streptococcus, and E_1, E_2, and E_3 correspond to the three pieces of joint evidence given in the IF part, respectively.

MYCIN uses measures of both belief and disbelief to represent degrees of confirmation and disconfirmation respectively in a given hypothesis. The basic measure of belief, denoted by $MB(H,E)$, is actually a measure of the *increased* belief in hypothesis H due to the evidence E. This is roughly equivalent to the estimated increase in probability of $P(H|E)$ over $P(H)$ given by an expert as a result of the knowledge gained by E. A value of 0 corresponds to no increase in belief and 1 corresponds to maximum increase or absolute belief. Likewise, $MD(H,E)$ is a measure of the increased disbelief in hypothesis H due to evidence E. MD ranges from 0 to $+1$, also, with $+1$ representing maximum increase in disbelief, (total disbelief) and 0 representing no increase. In both measures, the evidence E may be absent or may be replaced with another hypothesis, $MB(H_1,H_2)$. This represents the increased belief in H_1 given H_2 is true.

In an attempt to formalize the uncertainty measure in MYCIN, definitions of MB and MD have been given in terms of prior and conditional probabilities. It

should be remembered, however, the actual values are often subjective probability estimates provided by a physician. We have for the definitions

$$MB(H,E) = \begin{cases} 1 & \text{if } P(H) = 1 \\ \dfrac{max[P(H|E),P(H)] - P(H)}{max[1,0] - P(H)} & \text{otherwise} \end{cases} \quad (6.11)$$

$$MD(H,E) = \begin{cases} 1 & \text{if } P(H) = 0 \\ \dfrac{min[P(H|E),P(H)] - P(H)}{min[1,0] - P(H)} & \text{otherwise} \end{cases} \quad (6.12)$$

Note that when $0 < P(H) < 1$, and E and H are independent (so $P(H|E) = P(H)$), then $MB = MD = 0$. This would be the case if E provided no useful information.

The two measures MB and MD are combined into a single measure called the certainty factor (CF), defined by

$$CF(H,E) = MB(H,E) - MD(H,E) \quad (6.13)$$

Note that the value of CF ranges from -1 (certain disbelief) to $+1$ (certain belief). Furthermore, a value of $CF = 0$ will result if E neither confirms nor unconfirms H (E and H are independent).

In MYCIN, each rule hypothesis H_i has an associated MB and MD initially set to zero. As evidence is accumulated, they are updated using intermediate combining functions, and, when all applicable rules have been executed, a final CF is calculated for each H_i. These are then compared and the largest cumulative confirmations or disconfirmations are used to determine the appropriate therapy. A threshold value of $|CF| > 0.2$ is used to prevent the acceptance of a weakly supported hypothesis.

In the initial assignment of belief values an expert will consider all available confirming and disconfirming evidence, $E_1, \ldots, E_k$, and assign appropriate, consistent values to both. For example, in the assignment process, a value of 1 should be made if and only if a piece of evidence logically implies H ($\neg H$) with certainty. Additional rules related to the assignment process must also be carefully followed when using such methods.

Ad hoc methods have been used in a large number of knowledge-based systems, more so than have the more formal methods. This is largely because of the difficulties encountered in acquiring large numbers of reliable probabilities related to the given domain and to the complexities of the ensuing calculations. But, in bypassing the formal approaches one should question what end results can be expected. Are they poorer than would be obtained using formal methods? The answer to this question seems to be not likely. Sensitivity analyses (Buchanan et al., 1984) seem to indicate that the outcomes are not too sensitive to either the method nor the actual values used for many systems. However, much work remains to be done in this area before a useful theory can be formulated.

6.6 HEURISTIC REASONING METHODS

The approaches to uncertainty described so far seem to lack an AI flavor. The uncertainty in a given hypothesis is represented as a number which is combined with and compared to other numbers until a final number is translated into a weighted conclusion. Once the uncertainties have been translated into numbers, the cause of the uncertainty, its relative importance, necessity, and other factors are lost. Furthermore, this does not appear to be the process with which humans reason about uncertainty. Although we do weigh evidence both pro and con, we usually do not arrive at a net numeric estimate of a conclusion, only whether or not the conclusion is justified. In place of numbers, our reasoning appears to depend more on heuristics when reasoning with uncertain, incomplete knowledge. In this section we briefly consider this type of approach.

Heuristic methods are based on the use of procedures, rules, and other forms of encoded knowledge to achieve specified goals under uncertainty. Using both domain specific and general heuristics, one of several alternative conclusions may be chosen through the strength of positive versus negative evidence presented in the form of justifications or endorsements. The endorsement weights employed in such systems need not be numeric. Some form of ordering or preference selection scheme must be used, however.

For example, in a prototype system (named SOLOMON) developed by Paul Cohen (1985), endorsements in the form of domain and general heuristics are used to reason about uncertainties associated with a client's investment portfolio. In selecting investments for clients, there are many sources of uncertainty to contend with. First, there is uncertainty related to the client's lifestyle and financial status as well as his or her objectives. Secondly, there are varying degrees of uncertainty related to all types of investments. Some important factors which influence an investor's status include age, preretirement income expected, net worth, income needs, retirement programs, and tax bracket. Factors related to investments include corporate earnings, dividends, bond and money market yields, the direction of the market, and the rate of inflation, to name a few.

Endorsements in the SOLOMON system are justifications or reasons for believing or disbelieving a proposition or an inference. They are provided by experts as heuristics in place of numeric probability estimates. They are expressed as rules about the properties and relationships of domain objects. For example, heuristics used to reason about the uncertainties related to a client's status might take the form of rules such as

IF: Client income need is high and net worth is medium to high,
THEN: Risk-tolerance level is medium.

IF: Client tax bracket is high and risk-tolerance level is low,
THEN: Tax-exempt mutual-funds are indicated.

IF: Client age is high and income needs are high and retirement income is medium,

THEN: Risk-tolerance is low.

IF: Two positive endorsements are medium or high and one negative endorsement is high,

THEN: Then favor the positive choice.

Endorsements are used to control the reasoning process in at least two different ways. First, preference is given to rules which are strongly supported. Second, endorsements permit the condition or left-hand side of a rule to be satisfied (or rejected) without finding an exact match in the KB. The SOLOMON system is goal driven and uses a form of backward rule chaining. A goal is achieved when all of the inference conditions in the left-hand side of the goal rule have been proved. This requires proving subgoals and sub-subgoals until the chain of inferences is completed.

The control structure in SOLOMON is based on the use of an agenda where tasks derived from rules are ordered for completion on the strength of their endorsements. Strongly endorsed tasks are scheduled ahead of weakly endorsed ones. And when a task is removed for execution, endorsements are checked to see if they are still worth completing.

Endorsements are propagated over inferences $P \rightarrow Q$ by combining, replacing, or dropping endorsements E_P associated with antecedents P, endorsements of the implication itself, and other evidential relationships between Q and conclusions in the KB.

The SOLOMON system borrowed several design features from another heuristic reasoning system developed by Douglas Lenat called AM (Davis and Lenat, 1982). AM discovers basic concepts in mathematics by investigating examples of a newly generated conjecture and looking for regularities and extreme or boundary values in the examples. With an exponential number of available tasks, the system is always uncertain about what to work on next. For this, AM also uses an agenda to schedule its tasks for further investigation. Here again, heuristics are used to control the reasoning process. The system does this by developing a numerical "interest" factor rating for tasks which is used to determine the task's position on the agenda. Like the SOLOMON system, AM gives more strength to rules which have supportive evidence.

Although both AM and SOLOMON take into account the importance of the evidence, SOLOMON differs in one respect. SOLOMON also accounts for the accuracy of the evidence just as the testimony of an eyewitness is more convincing than circumstantial evidence. AM is unable to assess accuracy as such.

6.7 SUMMARY

Commonsense reasoning is nonmonotonic in nature. It requires the ability to reason with incomplete, and uncertain knowledge. Many AI systems model uncertainty with probabilities or levels of confidence and probability inference computations.

Each belief is assigned a degree of truthfulness or plausibility. Evidence is pooled through combining functions which compute a combined belief probability. Two popular approaches were considered in this chapter: the Bayesian probability method and the Dempster-Shafer approach.

The Bayesian approach depends on the use of known-prior and likely probabilities to compute conditional probabilities. The Dempster-Shafer approach on the other hand is a generalization of classical probability theory which permits the assignment of probability masses (beliefs) to all subsets of the universe and not just to the basic elements. A measure of belief for an assertion is then computed as a subinterval of [0,1], where the length of the interval measures the uncertainty.

In addition to methods based on formal theories the more pragmatic ad hoc approaches to uncertain reasoning were examined. In particular, the procedures used in MYCIN which combine measures of belief and disbelief into certainty factors were described. Because of its success, a number of expert systems designs have been patterned after this method.

Finally, heuristic, nonnumeric approaches to the uncertainty problem were considered. Here, endorsements for a given alternative would outweigh negative factors if general rules, data, and other domain knowledge provided stronger support. The SOLOMON and AM systems use a form of heuristic control to reason with uncertain knowledge.

EXERCISES

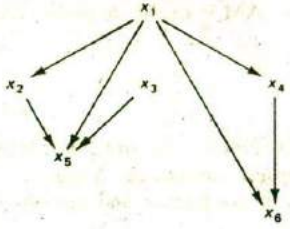
6.1 Find the probability of the event A when it is known that some event B occurred. From experiments it has been determined that $P(B|A) = 0.84$, $P(A) = 0.2$, and $P(B) = 0.34$.

6.2 Prove that if A and B are independent, $P(A|B) = P(A)$. (Note that A and B are independent if and only if $P(A \ \& \ B) = P(A)P(B)$).

6.3 From basic set theory prove that $P(\neg A) = 1 - P(A)$, and that $P(\neg B|A) = 1 - P(B|A)$.

6.4 Is it possible to compute $P(A|\neg B)$ when you are only given $P(A)$, $P(B|A)$, and $P(B)$? Explain your answer.

6.5 Write the joint distribution of x_1, x_2, x_3, x_4, x_5, and x_6 as a product of the chain conditional probabilities for the following causal network:



6.6 Define the sentences S_1, S_2, and S_3 as $S_1 = P$, $S_2 = Q$, and $S_3 = P \rightarrow Q$. Determine the probabilistic truth values of S_1, S_2, and S_3 when it is known that the probabilities of the possible worlds are given by $P(W_1) = 1/4$, $P(W_2) = 1/8$, $P(W_3) = 1/8$, and $P(W_4) = 1/2$.

6.7 Write a LISP program that computes certainty factors for rules based on the ad hoc MYCIN model.

6.8 Dempster-Shafer computations were given for four terrorist organizations A, B, C, and D in Section 6.4. Suppose now that new evidence (m_3) indicates that organization C was indeed responsible to a degree of 0.8. This requires that values for $m_3 \oplus m_4$ be computed, where $m_4 = m_1 + m_2$. Compute a new intersection tableau for the new evidence, that is compute $m_3(C)$ and $m_3(U)$ versus $m_4(A)$, $m_4(C,A)$, $m_4(A,B,D)$, and $m_4(U)$.

6.9 In what ways do endorsement justifications differ from probabilistic justifications?

6.10 In what ways do endorsement justifications differ from fuzzy logic justifications?

7

Structured Knowledge: Graphs, Frames, and Related Structures

The representations considered up to this point focused primarily on expressiveness, validity, consistency, inference methods, and related topics. Little consideration was given to the way in which the knowledge was structured and how it might be viewed by designers, or to the type of data structures that should be used internally. Neither was much consideration given to effective methods of organizing the knowledge structures in memory. In this, and the following two chapters, we address such problems.

In this chapter, we first look at associative networks as a form of representation and see examples of their use and flexibility. Next, we look at frames, a generalized structure recently introduced as a method of grouping and linking related chunks of knowledge. Finally, we consider structures closely related to frames known as scripts and memory organization packets. We compare them with the frame structures and see how they are used in reasoning systems.

7.1 INTRODUCTION

The representations studied in Chapter 4 are suitable for the expression of fairly simple facts. They can be written in clausal form as independent units and placed in a KB in any order. Inference is straightforward with a procedure such as chaining

```
PROFESSION(bob,professor)
FACULTY(bob,engineering)

MARRIED(bob,sandy)
FATHER-OF(bob,sue,joe)
DRIVES(bob,buick)
OWNS(bob,house)
~MARRIED(x,y) V MARRIED(y,x)
```

Figure 7.1 Facts in a KB given in clausal form.

or resolution. For example, facts about Bob, a university professor, might be entered as clauses in a KB as depicted in Figure 7.1.

The entries in the KB of Figure 7.1 have no particular order or grouping associated with them. Furthermore, in representing various facts about Bob, it was necessary to repeat Bob's name for each association given. All facts appear independently, without any linkage to other facts, even though they may be closely related conceptually (Bob is married, owns a house, has children, drives a Buick, and so forth).

For small KBs, the representation used in Figure 7.1 presents no problem. Adding, or otherwise changing facts in the KB is easy enough, and a search of all clauses in the KB can be made if necessary when performing inferences. When the quantity of information becomes large and more complex, however, the acquisition, comprehension, use, and maintenance of the knowledge can become difficult or even intractible. In such cases, some form of knowledge structuring and organization becomes a necessity.

Real-world problem domains typically involve a number and variety of different objects interacting with each other in different ways. The objects themselves may require extensive characterizations, and their interaction relationships with other objects may be very complex.

7.2 ASSOCIATIVE NETWORKS

Network representations provide a means of structuring and exhibiting the structure in knowledge. In a network, pieces of knowledge are clustered together into coherent semantic groups. Networks also provide a more natural way to map to and from natural language than do other representation schemes. Network representations give a pictorial presentation of objects, their attributes and the relationships that exist between them and other entities. In this section, we describe general associative

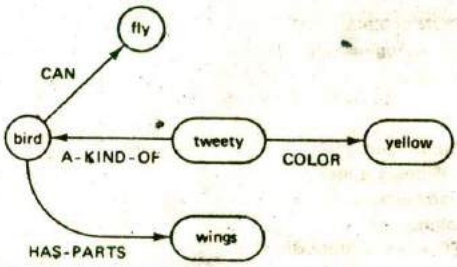


Figure 7.2 Fragment of an associative network.

networks (also known as semantic networks) and conceptual graphs and give some of their properties.

Associative networks are directed graphs with labeled nodes and arcs or arrows. The language used in constructing a network is based on selected domain primitives for objects and relations as well as some general primitives. A fragment of a simple network is illustrated in Figure 7.2. In the figure, a class of objects known as Bird is depicted. The class has some properties and a specific member of the class named Tweety is shown. The color of Tweety is seen to be yellow.

Associative networks were introduced by Quillian (1968) to model the semantics of English sentences and words. He called his structures semantic networks to signify their intended use. He developed a system which "found" the meanings between words by the paths connecting them. The connections were determined through a kind of "spreading activation" between the two words.

Quillian's model of semantic networks has a certain intuitive appeal in that related information is clustered and bound together through relational links. The knowledge required for the performance of some task is typically contained within a narrow domain or "semantic vicinity" of the task. This type of organization in some way, resembles the way knowledge is stored and retrieved in humans.

The graphical portrayal of knowledge can also be somewhat more expressive than other representation schemes. This probably accounts for the popularity and the diversity of representation models for which they have been employed. They have, for example, been used in a variety of systems such as natural language understanding, information retrieval, deductive data bases, learning systems, computer vision, and in speech generation systems.

Syntax and Semantics of Associative Networks

Unlike FOPL, there is no generally accepted syntax nor semantics for associative networks. Such rules tend to be designer dependent and vary greatly from one implementation to another. Most network systems are based on PL or FOPL with extensions, however. The syntax for any given system is determined by the object and relation primitives chosen and by any special rules used to connect nodes. Some efforts have been made toward the establishment of acceptable standards by

Schubert, Goebel, and Cercone (1979), Shapiro (1979), Hendrix (1979), and Brachman (1979). Later in this section we will review one formal approach to graphical representations which was recently proposed by John Sowa (1984).

Basically, the language of associative networks is formed from letters of the alphabet, both upper- and lowercase, relational symbols, set membership and subset symbols, decimal digits, square and oval nodes, and directed arcs of arbitrary length. The word symbols used are those which represent object constants and n-ary relation constants. Nodes are commonly used for objects or nouns, and arcs (or arc nodes) for relations. The direction of an arc is usually taken from the first to subsequent arguments as they appear in a relational statement. Thus, OWNS(bobs,house) would be written as

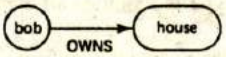


Figure 7.3 depicts graphically some additional concepts not expressed in Figure 7.1.

A number of arc relations have become common among users. They include such predicates as ISA, MEMBER-OF, SUBSET-OF, AKO (a-kind-of), HAS-

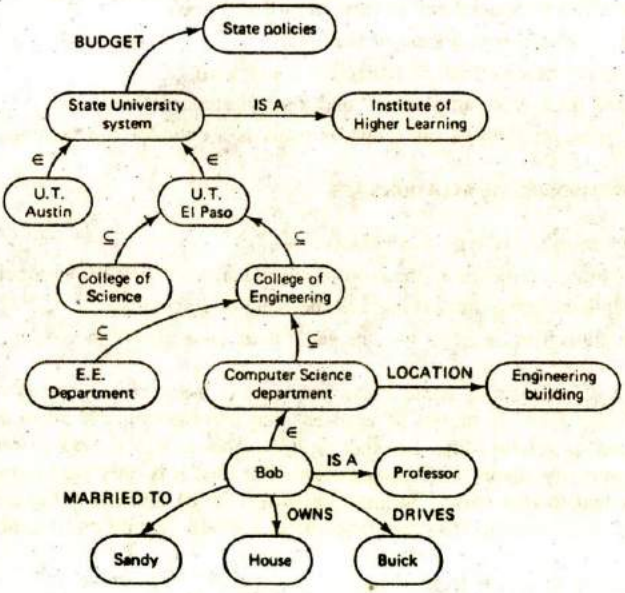


Figure 7.3 Associative network node and arc types.

PARTS, INSTANCE-OF, AGENT, ATTRIBUTES, SHAPED-LIKE, and so forth. Less common arcs have also been used to express modality relations (time, manner, mood), linguistics case relations (theme, source, goal), logical connectives (or, not, and, implies), quantifiers (all, some), set relations (superset, subset, member), attributes, and quantification (ordinal, count).

One particular arc or link, the ISA (is a) link, has taken on a special meaning. It signifies that Bob is a professor and that the state university system is an institute of higher learning. ISA relationships occur in many representations of worlds: Bill is a student, a cat is a furry animal, a tree is a plant, and so on. The ISA link is most often used to represent the fact that an object is of a certain type (predication) or to express the fact that one type is a subtype of another (for example, conditional quantification).

Brachman (1979, 1983) has given an interesting description of the background and uses of this now famous link. For example, the ISA predicate has been used to exhibit the following types of structures:

GENERIC-GENERIC RELATIONSHIPS

Subset-Superset (fighting ships-battleships)
Generalization-Specialization (restaurant-fast-foods)
AKO (an elephant is a kind of mammal)
Conceptual containment (a triangle is a polygon)
Sets and their type (an elephant and a set of elephants)
Role value restrictions (an elephant trunk is a cylinder 1.3 meters in length)

GENERIC-INDIVIDUAL RELATIONSHIPS

Set membership (Clyde is a camel)
Predication (predicate application to individual as in BROWN(camel))
Conceptual containment (king and the King of England)
Abstraction (the "eagle" in "the eagle is an endangered species")

Figure 7.3 illustrates some important features associative networks are good at representing. First, it should be apparent that networks clearly show an entity's attributes and its relationships to other entities. This makes it easy to retrieve the properties an entity shares with other entities. For this, it is only necessary to check direct links tied to that entity. Second, networks can be constructed to exhibit any hierarchical or taxonomic structure inherent in a group of entities or concepts. For example, at the top of the structure in Figure 7.3 is the Texas State University System. One level down from this node are specific state universities within the system. One of these universities, the University of Texas at El Paso, is shown with some of its subparts, colleges, which in turn have subparts, the different departments. One member of the Computer Science Department is Bob, a professor who

owns a house and is married to Sandy. Finally, we see that networks depict the way in which knowledge is distributed or clustered about entities in a KB.

Associative network structures permit the implementation of property inheritance, a form of inference. Nodes which are members or subsets of other nodes may inherit properties from their higher level ancester nodes. For example, from the network of Figure 7.4, it is possible to infer that a mouse has hair and drinks milk.

Property inheritance of this type is recognized as a form of default reasoning. The assumption is made that unless there is information to the contrary, it is reasonable for an entity to inherit characteristics from its ancestor nodes. As the name suggests, this type of inheritance is called default inheritance. When an object does not or cannot inherit certain properties, it would be assigned values of its own which override any inherited ones.

Data structures patterned after associative networks also permit the efficient storage of information since it is only necessary to explicitly store objects and shared properties once. Shared properties are attached only to the highest node in a structure to which they apply. For example, in LISP, Bob's associations (Figure 7.3) can be implemented with property lists where all of Bob's properties are linked to one atom.

```
(putprop 'bob 'cs-dept 'member-of)
(putprop 'bob 'professor 'isa)
(putprop 'bob 'sandy 'married-to)
(putprop 'bob 'house 'owns)
(putprop 'bob 'buick 'drives)
```

The semantics of associative networks are sometimes defined along the same lines as that of traditional logics. In fact, some network system definitions provide a means of mapping to and from PL or FOPL expressions. For these systems, the semantics are based on interpretations. Thus, an interpretation satisfies a portion of a network if and only if all arc relations hold in the given portion.

Inference procedures for networks can also parallel those of PL and FOPL. If a class A of objects has some property P, and a is a member of A, we infer that a has property P. Syntactic inference in networks can also be defined using parallels to traditional logics such as unification, chaining, modus ponens, and even resolution.

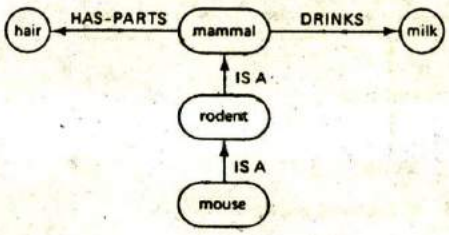


Figure 7.4 Property inheritance in a heirarchical network.

These procedures are implemented through node and arc matching processes and operators which insert, erase, copy, simplify, and join networks. We examine some typical inferencing procedures in more detail below.

Conceptual Graphs

Although there are no commonly accepted standards for a syntax and semantics for associative networks, we present an approach in this section which we feel may at least become a de facto standard in the future. It is based on the use of the conceptual graph as a primitive building block for associative networks. The formalism of these graphs has been adopted as a basic representation by a number of AI researchers and a variety of implementations using conceptual graphs are currently under development. Much of the popularity of these graphs has been due to recent work by John Sowa (1984) and his colleagues.

A conceptual graph is a graphical portrayal of a mental perception which consists of basic or primitive concepts and the relationships that exist between the concepts. A single conceptual graph is roughly equivalent to a graphical diagram of a natural language sentence where the words are depicted as concepts and relationships. Conceptual graphs may be regarded as formal building blocks for associative networks which, when linked together in a coherent way, form a more complex knowledge structure. An example of such a graph which represents the sentence "Joe is eating soup with a spoon" is depicted in Figure 7.5.

In Figure 7.5, concepts are enclosed in boxes and relations between the concepts are enclosed in ovals. The direction of the arrow corresponds to the order of the arguments in the relation they connect. The last or *n*th arc (argument) points away from the circle relation and all other arcs point toward the relation.

Concept symbols refer to entities, actions, properties, or events in the world. A concept may be individual or generic. Individual concepts have a type field followed by a referrent field. The concept [PERSON:joe] has type PERSON and referrent Joe. Referrents like joe and food in Figure 7.5 are called individual concepts since they refer to specific entities. EAT and SPOON have no referrent fields since they are generic concepts which refer to unspecified entities. Concepts like AGENT, OBJECT, INSTRUMENT, and PART are obtained from a collection of standard concepts. New concepts and relations can also be defined from these basic ones.

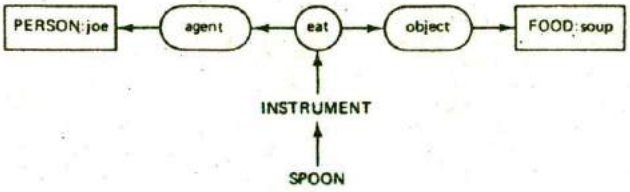


Figure 7.5 A conceptual graph.

A linear conceptual graph form which is easier to present as text can also be given. The linear form equivalent to the above sentence is

```
[PERSON:joe] ←(AGENT) ←[EAT]-
                        (OBJECT) →[FOOD:soup]
                        (INSTRUMENT) →[SPOON]
```

where square brackets have replaced concept boxes and parentheses have replaced relation circles.

The language of conceptual graphs is formed from upper- and lowercase letters of the alphabet, boxes and circles, directed arcs, and a number of special characters including -, ?, !, *, #, @, ∀, ~, ", :, [,], (,), →, ←, {, and }. Some symbols are used to exhibit the structure of the graph, while others are used to determine the referrents.

The dash signifies continuation of the linear graph on the next line. The question mark is used to signify a query about a concept when placed in the referrent field: [HOUSE:?] means which house? The exclamation mark is used for emphasis to draw attention to a concept. The asterisk signifies a variable or unspecified object: [HOUSE:*x] means a house or some house. The pound sign signifies a definite article known to the speaker. For example, [House:#432] refers to a specific house, house number 432. The @ symbol relates to quantification: [HOUSE:@ n] means *n* houses. ∀ signifies every or all, the same as in FOPL. The tilde is negation. Double quotation marks delimit literal strings. And the colon, brackets, parentheses, and directed arcs are used to construct graph structures as illustrated above.

Since conceptual graphs and FOPL are both a form of logical system, one might expect that it is possible to map from one representation to the other. Indeed this is the case, although some mappings will, in general, result in second order FOPL statements.

To transform a conceptual graph to a predicate logic statement requires that unique variable names be assigned to every generic concept of the graph. Thus, the concepts EAT and FOOD of Figure 7.5 would be assigned the variable names *x* and *y*, respectively. Next, all type labels such as PERSON and FOOD are converted to unary predicates with the same name. Conceptual relations such as AGENT, OBJECT, and INSTRUMENT are converted to predicates with as many arguments as there are arcs connected to the relation. Concept referrents such as Joe and soup become FOPL constants. Concepts with extended referrents such as ∀ map to the universal quantifier ∀. Generic concepts with no quantifier in the referrent field have an existential quantifier, ∃, placed before the formula for each variable, and conjunction symbols, &, are placed between the predicates.

As an example, one could convert the sentence "Every car has an engine" from its conceptual graph representation given by

```
[CAR:∀] →(PART) →[ENGINE]
```

to its equivalent FOPL representation. Using the rules outlined above, the equivalent FOPL representation derived is just

$$\forall x\ \exists y\ (CAR(x) \rightarrow (ENGINE(y)\ \&\ PART(x,y)))$$

Mapping the other way, that is from FOPL statements to conceptual graphs, begins by putting the FOPL formula into prenex normal form, and converting all logical connectives to negation and conjunction. Next, every occurrence of universal quantification $\forall x$ is replaced with the equivalent form $\neg\exists\neg x$ (in graph notation this is $\neg[\ ]x\neg[$ with the subsequent addition of balancing brackets $]\ ]$ to close off the expression). Every variable x and every occurrence of $\exists x$ is then replaced with the most general type concept denoted as [T:*x]. And finally, every n-ary predicate symbol is replaced with an n-ary concept relation whose ith arc is attached to the concept in the ith argument place in the predicate.

Implication in a conceptual graph can be represented with negation and conjunction. For example, the FOPL equivalent of $P \rightarrow Q$ can be written as $\neg[P\ \neg[Q]]$ (recall that $P \rightarrow Q = (\neg P\ \vee\ Q) = \neg(P\ \&\ \neg Q)$). In this expression, $\neg[$ is read *as if* and the nested $\neg[$ is read *as then*. More generally, we write the implication as $\neg[*p\neg[*q]]$ where $*p$ and $*q$ are themselves any conceptual graph.

Inference can be accomplished by modifying and combining graphs through the use of operators and basic graph inference rules. Four useful graph formation operators are copy, restrict, join, and simplify. These operators are defined as follows.

Copy. Produces a duplicate copy of a CG.

Restrict. Modifies a graph by replacing a type label of a concept with a subtype or a specialization from generic to individual by inserting a referrent of the same concept type.

Join. Combines two identical graphs C_1 and C_2 by attaching all relation arcs from C_2 to C_1 and then erasing C_2.

Simplify. Eliminates one of two identical relations in a conceptual graph when all connecting arcs are also the same.

As an example of the use of the formation rules, consider the sentence "Tweety ate a fat worm." This sentence can be broken down into five basic or cannonical conceptual graphs corresponding to the five words in the sentence.

Tweety:	[BIRD:tweety]
ate:	[Animal] ← (AGENT) ← [ATE] → (Patient) → [ENTITY]
a:	[T: *]
fat:	[FAT] ← (ATTRIBUTE) ← [PHYSICAL-OBJECT]
worm:	[WORM]

The [T:*] signifies that something of an unspecified type exists (T is the most general type of all concepts).

From these basic graphs a single conceptual graph can be constructed using the formation operators. First, the subgraph from "a fat worm" is constructed by *restricting* PHYSICAL-OBJECT in the fat graph to WORM and then *joining* it to the graph for worm to get [FAT] ← [ATTRIBUTE] ← [WORM]. Next, ENTITY in the graph for ate is *restricted* to WORM and *joined* to the graph just completed. This gives

[ANIMAL] ← (AGENT) ← [ATE] → (PATIENT) → [WORM] ← (ATTRIBUTE) → [FAT].

The final conceptual graph is obtained by *restricting* ANIMAL to BIRD with referrent Tweety, *joining* the graphs and labeling the whole graph with PAST (for past tense).

(PAST) → [[BIRD: tweety] ← (AGENT) ← [EAT] → (PATIENT) – [WORM] → (ATTRIBUTE) → [FAT]]

In forming the above graph, concept specialization occurred (e.g., when restriction took place as in PHYSICAL-OBJECT to WORM). Thus, the formation rules and their inverses provide one method of inference. When rules for handling negation and other basic inference rules are combined with the formation rules, a complete inference system is obtained. This system is truth preserving. The inference rules needed which are the equivalent of those in a PL system are defined as follows.

Erasure. Any conceptual graph enclosed by an even number of negations may be erased.

Insertion. Any conceptual graph may be inserted into another graph context which is enclosed by an odd number of negations.

Iteration. A copy of any conceptual graph C may be inserted into a graph context in which C occurs or in which C is dominated by another concept.

Deiteration. Any conceptual graph which could be the result of iteration may be erased from a conceptual graph context.

Double Negation. A double negation may be erased or drawn before any conceptual graph or set of graphs.

As an example of some of the above rules, any graph w may be inserted in the implication $\neg[u\ \neg[v]]$ to derive $\neg[u\ w\ \neg[v]]$. Any graph w may be erased from the consequent of an implication $\neg[u\ \neg[v\ w]]$ to derive $\neg[u\ \neg[v]]$. Any graph w may be erased from the antecedent of an implication only if it has been independently asserted. Thus if u and $\neg[u\ v\ \neg[w]]$ then derive $\neg[v\ \neg w]]$.

Note that deiteration and double negation are equivalent to modus ponens, that is, given p and $\neg[p\ \neg[q]]$, deiteration permits erasure of p inside the first bracket to get $\neg[\neg[q]]$. Double negation then permits erasure of $\neg[\neg$ to obtain the final result q.

Other inference methods including inheritance (if all *A* have property *P* and all *B* are *A*, all *B* have property *P*) and default reasoning are also possible with conceptual graphs. The implementation of modal logic formalisms with these graphs is possible by using concepts such as possible and necessary. Heuristic reasoning can be accomplished within the theory of conceptual graphs.

In summary, conceptual graphs offer the means to represent natural language statements accurately and to perform many forms of inference found in common sense reasoning.

7.3 FRAME STRUCTURES

Frames were first introduced by Marvin Minsky (1975) as a data structure to represent a mental model of a stereotypical situation such as driving a car, attending a meeting, or eating in a restaurant. Knowledge about an object or event is stored together in memory as a unit. Then, when a new situation is encountered, an appropriate frame is selected from memory for use in reasoning about the situation.

Frames are general record-like structures which consist of a collection of slots and slot values. The slots may be of any size and type. Slots typically have names and values or subfields called facets. Facets may also have names and any number of values. An example of a simple frame for Bob is depicted in Figure 7.6 and a general frame template structure is illustrated in Figure 7.7.

From Figure 7.7 it will be seen that a frame may have any number of slots, and a slot may have any number of facets, each with any number of values. This gives a very general framework from which to build a variety of knowledge structures.

The slots in a frame specify general or specific characteristics of the entity for which the frame represents, and sometimes they include instructions on how to apply or use the slot values. Typically, a slot contains information such as attribute value pairs, default values, conditions for filling a slot, pointers to other related frames, and procedures that are activated when needed for different purposes. For example, the Ford frame illustrated in Figure 7.8 has attribute-value slots (COLOR: silver, MODEL: 4-door, and the like), a slot which takes default values for GAS-MILEAGE, and a slot with an attached *if-needed* procedure.

```
(bob
      (PROFESSION (VALUE professor))
      (AGE (VALUE 42))
      (WIFE (VALUE sandy))
      (CHILDREN (VALUE sue joe))
      (ADDRESS (STREET (VALUE 100 elm))
               (CITY (VALUE dallas))
               (STATE (VALUE tx))
               (ZIP (VALUE 75000))))
```

Figure 7.6 A simple instantiated person frame.

```
(<frame name>
    (<slot1> (<facet1><value1>....<valuek1>)
             (<facet2><value1>....<valuek2>)

    (< slot2> (<facet1><value1>....<valuekm>)
```

Figure 7.7 A general frame structure.

The value fget in the GAS-MILEAGE slot is a function call to fetch a default value from another frame such as the general car frame for which Ford is a-kind-of (AKO). When the value of this slot is evaluated, the fget function is activated. When fget finds no value for gas mileage it recursively looks for a value from ancestor frames until a value is found.

The if-needed value in the Range slot is a procedure name that, when called, computes the driving range of the Ford as a function of gas mileage and fuel capacity. Slots with attached procedures such as fget and if-needed are called procedural attachments or demons. They are done automatically when a value is needed but not provided for in a slot. Other types of demons include *if-added* and *if-removed* procedures. They would be triggered, for example, when a value is added or removed from a slot and other actions are needed such as updating slots in other dependent frames.

Like associative networks, frames are usually linked together in a network through the use of special pointers such as the AKO pointer in Figure 7.8. Hierarchies of frames are typical for many systems where entities have supertype-subtype or generic-instance relationships. Such networks make it possible to easily implement property inheritance and default reasoning as depicted in Figure 7.8. This is illustrated

```
(ford (AKO (VALUE car))
      (COLOR (VALUE silver))
      (MODEL (VALUE 4-door))
      (GAS-MILEAGE (DEFAULT fget))
      (RANGE (VALUE if-needed))
      (WEIGHT (VALUE 2600))
      (FUEL-CAPACITY (VALUE 18))
```

Figure 7.8 A Ford frame with various slot types.

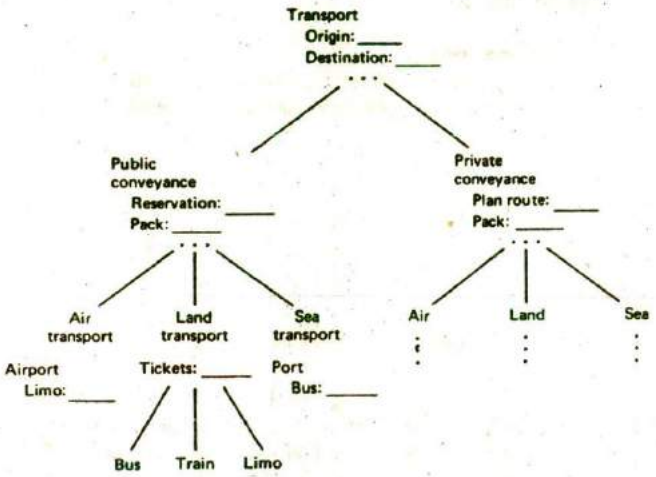


Figure 7.9 Network of frames for transportation methods.

in the network of frames which represents various forms of transportation for people (Figure 7.9).

Frame-Based Representation Languages

Frame representations have become popular enough that special high level frame-based representation languages have been developed. Most of these languages use LISP as the host language. They typically have functions to create, access, modify, update, and display frames. For example, a function which defines a frame might be called with

```
(fdefine f-name <parents><slots>)
```

where fdefine is a frame definition function, f-name is the name assigned to the new frame, <parents> is a list of all parent frames to which the new frame is linked, and <slots> is a list of slot names and initial values. Using the function fdefine to create a train frame we might provide the following details.

```
(fdefine general-train land-transport
   (type (VALUE passenger))
   (class (VALUE first-class second-class sleeper))
   (food (restaurant (VALUE hot-meals))
         (fast-food (VALUE cold-snacks))
   ...
```

A few other functions typically provided in a frame language include

```
(fget f-name slot-name facet-name)        ;returns data from
                                          ;specified location
(fslots f-name)                           ;returns names of
                                          ;slots
(ffacets f-name slot-name)                ;returns names of
                                          ;facets
(fput f-name slot-name facet-name)        ;adds data to a
                                          ;specified location
(fremove f-name slot-name facet-name)     ;removes data from
                                          ;specified location
```

Several frame languages have now been developed to aid in building frame-based systems. They include the Frame Representation Language (FRL) (Bobrow et al., 1977), Knowledge Representation Language (KRL), which served as a base language for a scheduling system called NUDGE (Goldstein et al., 1977) and KLONE (Brachman, 1978).

Implementation of Frame Structures

One way to implement frames is with property lists. An atom is used as the frame name and slots are given as properties. Facets and values within slots become lists of lists for the slot property. For example, to represent a train frame we define the following putprop.

```
(putprop 'train ((type (VALUE passenger))
                 (class(VALUE first second sleeper))
                 (food (restaurant (VALUE hot-meals))
                       (fast-food (VALUE cold snacks)))
                 'land-transport)
```

Another way to implement frames is with an association list (an a-list), that is, a list of sublists where each sublist contains a key and one or more corresponding values. The same train frame would be represented using an a-list as

```
(setq train '((AKO land-transport)
              (type (VALUE passenger))
              (class(VALUE first second sleeper))
              (food (restaurant (VALUE hot-meals))
                    (fast-food (VALUE cold snacks)))))
```

It is also possible to represent frame-like structures using object-oriented programming extensions to LISP languages such as FLAVORS (described in Chapter 8).

7.4 CONCEPTUAL DEPENDENCIES AND SCRIPTS

Scripts are another structured representation scheme introduced by Roger Schank (1977). They are used to represent sequences of commonly occurring events. They were originally developed to capture the meanings of stories or to "understand" natural language text. In that respect they are like a script for a play.

A script is a predefined frame-like structure which contains expectations, inferences, and other knowledge that is relevant to a stereotypical situation. Scripts are constructed using basic primitive concepts and rules of formation somewhat like the conceptual graphs described in Section 7.2. Before proceeding with a description of the script, we describe the primitives and related rules used in building them. They are known as conceptual dependencies (not to be confused with conceptual graphs).

Conceptual Dependencies

Conceptual dependency (CD) theory is based on the use of a limited number of primitive concepts and rules of formation to represent any natural language statement. The theory states that different sentences which have the same meaning should have the same unique CD representation. Furthermore, representations for any sentence should be unambiguous as the speaker intends, even though it may have syntactic ambiguity, as in "I saw the Golden Gate Bridge flying into San Francisco this afternoon." It is the contention that any concept, like dreaming, thinking, bellowing, or scheming can be described in terms of these primitives.

In CD theory five different types of ontological (state of being) building blocks are distinguished. Each of these types, in turn, has several subtypes. The types are made up of entities, actions, conceptual cases, conceptual dependencies, and conceptual tenses.

ENTITIES

Picture producers (PP) are actors or physical objects (including human memory) that perform different acts.

Picture aiders (PA) are supporting properties or attributes of producers.

ACTIONS

Primitive actions (ACTs) as listed in Figure 7.10.

Action aiders (AA) are properties or attributes of primitive actions.

Primitive actions	Intended meaning
ATRANS	Transfer of an abstract entity
ATTEND	Focusing attention on an object
CONC	To think about something
EXPEL	Expulsion of anything from the body
GRASP	Grasping or holding an object tightly
INGEST	Ingesting something
MBUILD	Building on information
MOVE	Moving a part of the body
MTRANS	Transfer of mental information
PROPEL	Application of force
PTRANS	Physical transfer from one location to another
SPEAK	Emitting a sound

Figure 7.10 Conceptual dependency primitive actions.

CONCEPTUAL CASES (ALL ACTIONS INVOLVE ONE OR MORE OF THESE)

Objective Case
Directive Case
Instrumental Case
Recipient Case

CONCEPTUAL DEPENDENCIES

Semantic rules for the formation of dependency structures such as the relationship between an actor and an event or between a primitive action and an instrument. (see Figure 7.11).

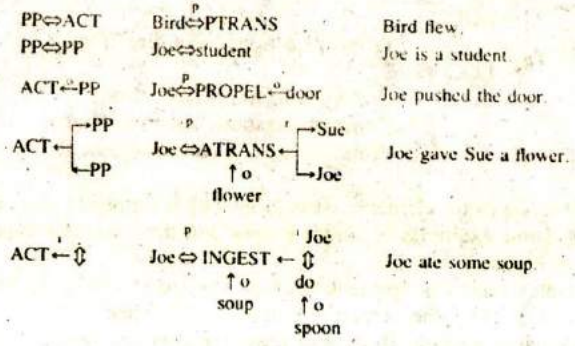


Figure 7.11 Some typical conceptual dependency structures.

CONCEPTUAL TENSES (TIME OF ACTION OR STATE OF BEING)

Conditional (c)
Continuing (k)
Finished Transition (tf)
Future (f)
Interrogative (?)
Negative (/)
Past (p)
Present (nil)
Start Transition (ts)
Timeless (delta)
Transition (t)

Conceptual structures in the form of a graph are used to represent the meanings of different English (or other language) sentences. The graphs are constructed from elementary structures in accordance with basic syntax rules. Some of the basic concept rules are as follows.

PP ACT	Some picture producers perform primitive actions.
PP PA	Picture producers have attributes.
ACT ← O — PP	Primitive actions have objects.
ACT ← R — (→ PP, ← pp)	Primitive actions have recipients.
ACT ← D — (→ LOC, ← LOC)	Primitive actions have directions.
T	Conceptualizations have times.
LOC	Conceptualizations have locations.
ACT ← I — ⇕	Primitive actions have instruments.

Using these syntactic elements, structures which represent any sentence can be constructed. Some examples of simple graphs and their corresponding sentences are illustrated in Figure 7.11.

More complex sentence representations are constructed from the basic building blocks given above. Note the similarities between CD theory and the conceptual graphs of the previous section. Both have primitive concepts and relations defined, and both have a syntax for graphical representation. Conceptual graphs differ from CDs mainly in that the conceptual graph is logic based, whereas CD theory is mainly concerned with the semantics of events. We now turn to the event representation structure which uses CDs, the script.

Scripts

Scripts are frame-like structures used to represent commonly occurring experiences such as going to the movies, shopping in a supermarket, eating in a restaurant, or visiting a dentist. Like a script for a play, the script structure is described in terms

SCRIPT-NAME:		food market
TRACK	:	supermarket
ROLES	:	shopper
		deli attendant
		seafood attendant
		checkout clerk
		sacking clerk
		other shoppers
ENTRY		
CONDITIONS	:	shopper needs groceries
		food market open
PROPS	:	shopping cart
		display aisles
		market items
		checkout stands
		cashier
		money
SCENE1	:	Enter Market
		shopper PTRANS shopper into market
		shopper PTRANS shopping-cart to shopper
SCENE2	:	Shop For Items
		shopper MOVE shopper through aisles
		shopper ATTEND eyes to display items
		shopper PTRANS items to shopping cart
SCENE3	:	Check Out
		shopper MOVE shopper to checkout stand
		shopper WAIT shopper turn
		shopper ATTEND eyes to charges
		shopper ATRANS money to cashier
		sacker ATRANS bags to shopper
SCENE4	:	Exit Market
		shopper PTRANS shopper to exit-market
RESULTS	:	shopper has less money
		shopper has grocery items
		market has less grocery items
		market has more money

Figure 7.12 A supermarket script structure.

of actors, roles, props, and scenes. Slots in a script which correspond to parts of the event are filled with CD primitives as defined above. An example of a supermarket script is illustrated in Figure 7.12. This script has four scenes which correspond to the main events which commonly occur in a supermarket shopping experience.

Reasoning with Scripts

Since scripts contain knowledge that people use for common every day activities, they can be used to provide an expected scenario for a given situation.

Reasoning in a script begins with the creation of a partially filled script named to meet the current situation. Next, a known script which matches the current situation is recalled from memory. The script name, preconditions, or other key words provide index values with which to search for the appropriate script. Inference is accomplished by filling in slots with inherited and default values that satisfy certain conditions. For example, if it is known that Joe-PTRANS-Joe into a supermarket and Joe-ATRANS-cashier money, it can be inferred that Joe needed groceries, shopped for items, paid the cashier, checked out, and left the market with groceries but with less money than when he entered.

Scripts have now been used in a number of language understanding systems (English as well as other languages) at Yale University by Schank and his colleagues. One such system is SAM (Script Applier Mechanism) which reads and reasons with text to demonstrate an "understanding" of stories (such as car accident stories from newspapers) that were script based. Other programs developed at Yale include PAM, POLITICS, FRUMP, IPP, BORIS, BABEL, and CYRUS. All of these programs deal with reading, planning, explaining, or in some way understanding stories. They all used some form of script representation scheme.

7.5 SUMMARY

In this chapter we have investigated different types of structured knowledge representation methods. We first considered associative networks, a representation based on a structure of linked nodes (concepts) and arcs (relations) connecting the nodes. With these networks we saw how related concepts could be structured into cohesive units and exhibited as a graphical representation.

Next, we looked at conceptual graphs, a structured formalism based on traditional logics and which uses primitive building blocks for concepts and relationships between the concepts. How conceptual graphs and formulas in FOPL could be mapped from one to the other and how inferring with conceptual graphs compared to logical inference were demonstrated.

We then considered frame structures as general methods of representing units of knowledge. Frames are composed of any number of slots, each with any number of facets, and they in turn, contain any number of values. The contents of a slot may be attributes which characterize the frame entity, pointers to related frames,

procedures, or even other subframes. Inference with frames is accomplished through property inheritance, default values, embedded procedures, and the use of heuristics.

Finally, we described a special frame-like structure called a script. Scripts are used to represent stereotypical patterns for commonly occurring events. Conceptual dependency theory provides primitive building blocks for actions and states that occur within a script. Like a play, a script contains actors, roles, props, and scenes which combine to represent a familiar situation. Scripts have been used in a number of programs which read and "understand" language in the form of stories.

EXERCISES

7.1 Express the following concepts as an associative network structure with interconnected nodes and labeled arcs.

Company ABC is a software development company. Three departments within the company are Sales, Administration, and Programming. Joe is the manager of Programming. Bill and Sue are programmers. Sue is married to Sam. Sam is an editor for Prentice Hall. They have three children, and they live on Elm street. Sue wears glasses and is five feet four inches tall.

7.2 Write LISP expressions which represent the associative network of Problem 7.1.
- **a.** using property lists, and
- **b.** using a-lists.

7.3 Write PROLOG expressions which represent the associative network of Problem 7.1.

7.4 Transform the FOPL statements given below into equivalent conceptual graphs.
- **a.** $\forall x$ NORMAL(x) & GROWN(x)→WALK(x).
- **b.** $\forall x,y$ MARRIED(x,y) → MARRIED(y,x).
- **c.** $\forall x$ HASWINGS(x) & LAYSEGGS(x) → ISBIRD(x).

7.5 Transform the following conceptual graphs into equivalent FOPL statements.
- **a.** [PERSON:sue] ←(AGENT) ←[DRINK]-
  (OBJECT)→ [FOOD:milk]
  (INSTRUMENT)→ [GLASS]
- **b.** (PAST)→ [[CAMEL:clyde] ←(AGENT) ←[DRINK]→ (OBJECT)-
  [WATER]→ (ATTRIBUTE)→ [50-GALLONS]]

7.6 The original primitives of conceptual dependency theory developed by Schank fail to represent some important concepts directly. What additional primitives can you discover that would be useful?

7.7 Create a movie script similar to the supermarket script of Figure 7.11.

7.8 What are the main differences between scripts and frame structures?

7.9 Express the following sentences as conceptual dependency structures.
- **a.** Bill is a programmer.
- **b.** Sam gave Mary a box of candy.
- **c.** Charlie drove the pickup fast.

7.10 Create a frame network for terrestrial motor vehicles (cars, trucks, motorcycles) and give one complete frame in detail for cars which includes the slots for the main component

parts, their attributes, and relations between parts. Include an as-needed slot for the gas of each type mileage.

7.11 Write a LISP program to create a frame data structure which represents the car frame of Problem 7.10.

7.12 Compare the inference process using frames to that of inference in FOPL. Give examples of both.

8

Object-Oriented Representations

The previous chapter marked a departure from the approaches to knowledge representation of earlier chapters in that the methods there focused on adding structure to the knowledge. Structure was added by linking concepts through relations and clustering together all related knowledge about an object. In some cases, as with frames, procedures related to the knowledge were also attached to the knowledge cluster. The approach in grouping knowledge and related procedures together into a cohesive unit is carried even further with object-oriented systems which we examine in some detail in this chapter.

8.1 INTRODUCTION

Grouping related knowledge together in AI systems gains some of the same cognitive advantages realized in the human brain. The knowledge required for a given cognitive task is usually quite limited in domain and scope. Therefore, access and processing can be made more efficient by grouping or partitioning related knowledge together as an unit. We saw how this notion was implemented with linked frames in the previous chapter. We shall see in this chapter, that object-oriented systems share a number of similarities with the frame implementations.

In procedural programming languages such as Pascal or FORTRAN, a program

consists of a procedural part and a data part. The procedural part consists of the set of program instructions, and the data part, the numbers and character strings that are manipulated by the instructions. Programs typically contain several modules of instructions that perform computations on the same data set. When some change is made to the format of the data, every module that uses it must then be modified to accommodate the newly revised format. This places a heavy burden on the software maintenance process and makes these types of programs more prone to errors.

In an object-oriented system (OOS) the emphasis between data and procedures is reversed. Data becomes the primary object and procedures are secondary. For example, everything in the universe of an OOS is an object, and objects are inaccessible to outside procedures. This form of structuring is sometimes called encapsulation or data hiding. It is a well known system design principle used to make systems more modular and robust. With encapsulation, objects are associated with their own procedures and, as such, are responsible for their own actions. Thus, when some change is required in the data or procedure, only the changed object need be modified. Other objects are not affected and therefore require no modifications.

In object-oriented systems there is a simplicity in structure because almost everything is an object. For example, a car can be regarded as an object consisting of many interacting components or subobjects: an engine, electrical system, fuel system, drive train, controls, and so on. To model such a system using an object-oriented approach requires that all parts be declared as objects, each one characterized by its own attributes and its own operational behavior. Even a simple windshield wiper would be described as an object with given attributes and operations. As such, it might be described as the structure presented in Figure 8.1.

This object has a name, a class characterization, several distinguishing attributes, and a set of operations. Since all characteristics of the wiper object, including its operations, are contained within a single entity, only this entity needs changing when some design change is made to this part of the car. Other objects that interact with it are not affected, provided the communication procedures between the objects were not changed. To initiate the task of cleaning moisture from the windshield requires only that a message be sent to the object. This message can remain the

OBJECT NAME	left wiper
AKO	wiper
ATTRIBUTES	made of rubber and metal
	length: 14 inches
	color: black and silver
	location: lower left windshield
	function: rub moisture from windshield
OPERATIONS	turn-on switch: move in arc on windshield repeating clockwise then counter-clockwise
	turn-off switch: move to home position

Figure 8.1 Object description for a windshield wiper.

same even though the structure of the wiper or its mode of operation may have changed.

Because there is more than one wiper, each with similar attributes and operations, some savings in memory and procedures can be realized by creating a generic class which has all the characteristics which are common to the left, right, and rear wipers. The three instances retain some characteristics unique to themselves, but they inherit common attributes and operations from the more general wiper class.

The object paradigm described above seems to model real-world systems more closely than the procedural programming models where objects (data) and procedures are separated. In object-oriented systems, objects become individual, self-contained units that exhibit a certain behavior of their own and interact with other objects only through the passing of messages. Tasks get performed when a message is sent to an object that can perform the task. All the details of the task are rightfully hidden from other objects. For example, when your car needs repairing, you send a message to the repair shop. The repair shop, in turn, may need parts from one or more manufacturers for which they must send messages. When the car has been repaired, the repair shop sends you a message informing you of that fact.

In having the shop repair your car, you probably are not interested in all the details related to the repair, the fact that messages were sent to other organizations for parts, that they were obtained by Federal Express, and that certain detailed procedures were taken to complete the repair process. Your primary concern is that the repair operation was properly completed and your car returned in working order. The need to model operational behavior such as this has prompted the development of object-oriented systems.

2 OVERVIEW OF OBJECT-ORIENTED SYSTEMS

The basic idea behind an OOS is the notion of classes of objects interacting with each other to accomplish some set of tasks. The objects have well-defined behaviors. They interact with each other through the use of messages. When a task needs to be performed, an object is passed a message which specifies the task requirements. The receiving object then takes appropriate action in response to the message and responds by returning a message to the sender. In performing the required task, the receiver may need assistance from other objects, thereby prompting further messages to be sent.

These ideas are illustrated in Figure 8.2 which depicts the simulation of a seaport facility. Ocean ships arrive for docking, unloading, loading, and departing. When the facilities (tugboats, berths, and loading and unloading equipment and crews) are busy, arriving ships must queue and wait at sea until space and other facilities are available. The harbor master coordinates the arrivals and departures by assigning tugs and other resources to the arriving ships. The objects in this example are, of course, the individual ships, the tugs, the docks, the harbor master,

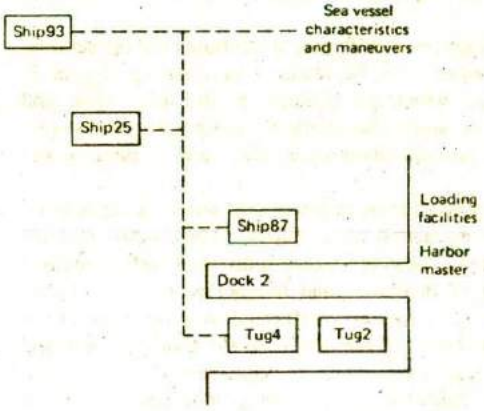


Figure 8.2 Objects communicating to complete a task.

and the cargo handling facilities. Actions are initiated by message passing between these objects. The dashed lines connecting members of the class of sea vessels depict the way common characteristics and operational behaviors are shared by members of the same class (they all have a coordinate position, a maximum cruising speed, cargo capacity, and so on).

Tasks are performed when a message is sent from one object to another. For exmaple, the harbor master may send a message to a tug to provide assistance to ship 87 in deberthing from dock 2. This would then result in a sequence of actions from the tug having received the message.

In general, a task may consist of any definable operation, such as changing an object's position, loading cargo, manipulating a character string, or popping up a prompt window. A complete program would then be a sequence of the basic tasks such as the simulated movement of ships into and out of the seaport after discharging and taking on cargo.

8.3 OBJECTS, CLASSES, MESSAGES, AND METHODS

In this section we present definitions for the basic concepts that make up an OOS: the object, message, class, methods, and class hierarchies. There are probably as many as fifty different OOS languages, and the examples presented in this section may not comply exactly with any one in particular. The examples are representative of all OOS however, and are based mainly on material from the Smalltalk family, including Smalltalk 80 (Goldberg and Robson, 1983, and Kaehler and Patterson, 1986), Smalltalk/V (Digitalk, Inc., 1986), and Little Smalltalk (Budd, 1987). Specialized OOS languages are considered in Section 8.5.

Objects

Objects are the basic building blocks in object-oriented systems. All entities except parts of a message, comments, and certain punctuation symbols are objects. An object consists of a limited amount of memory which contains other objects (data and procedures). They are encapsulated together as a unit and are accessible to that object only. Examples of objects are numbers such as 5, 31, 6.213, strings like 'this is a string,' arrays such as #(23 'a string' 311 (3 4 5)), the Turtle (a global graphics object originally used in LOGO), a windshield wiper as described above, a ship, and so on. Objects are characterized by attributes and by the way they behave when messages are sent to them. All objects belong to some class. They are created by declaring them as instances of an existing class and instantiating instance variables. The class to which an object belongs can be determined by sending it the message "class."

Messages

Actions are performed in an OOS by sending messages to an object. This corresponds to a function or procedure call in other languages. The messages are formatted strings composed of three parts: a receiver object, a message selector, and a sequence of zero or more arguments. The format of a message is given as

<object><selector><arg_1 arg_2 . . .>

The object identifies the receiver of the message. This field may contain an object item or another message which evaluates to an object. The selector is a procedure name. It specifies what action is required from the object. The arguments are objects used by the receiver object to complete some desired task. Messages may also be given in place of an argument since a message always elicits an object as a response.

When an object receives a valid message, it responds by taking appropriate actions (such as executing a procedure or sending messages to other objects) and then returning a result. For example, the message 9 - 5 causes the receiver object 9 to respond to the selector - by subtracting 5 from 9 and returning the object 4.

There are three types of messages: unary, binary, and keyword (*n*-ary). All three types parse from left to right, but parentheses may be used to determine the order of interpretation. A unary message requires no arguments. For example, each of the following are unary messages:

```
5 sign
10 factorial
'once upon a time' size
#(a b c d) reversed
68 asCharacter
```

In each of these examples, the first item in the message is the receiver object, and the second item the selector. The first example returns the integer +1 to signify a positive algebraic sign for the number 5. The second example returns 3628800 the factorial value of the integer 10. The third example returns 16, the length of the string. The fourth returns the array *#(d c b a)*, and the fifth returns *D*, the ASCII character equivalent of 68.

Binary messages take one argument. Arithmetic operations are typical of binary messages, where the first operand is the receiver, the selector is the arithmetic operation to be performed, and the second operand is the argument. Examples of binary messages are

```
10 + 32                 "an addition message"
13 - 9                  "a subtraction message"
22 * 7                  "multiplication message"
54 / 2                  "rational division message"
#(a b c), #(d e f)      "the comma concatenates two arrays"
7 < 9                   "relational test message"
7 @ 12                  "an x-y coordinate point reference"
```

Comments may be placed anywhere within an OOS program using the double quotation marks as seen in the above examples. Note that the last three examples are nonarithmetic binary messages. They result in the combining of two arrays into one, a boolean relational test, and the creation of a graphics coordinate point at column 7, row 12, respectively.

The third and most general type of message is the keyword message. These messages have selectors which consist of one or more keyword identifiers, where each is followed by a colon and an argument. The argument can be an object or any message, but if it is another keyword message, it must be enclosed in parentheses to avoid ambiguity. Examples of keyword messages are

```
5 between: 4 and: 10                "a Boolean test"

'aecdb' copyFrom: 2 to: 5           "copies position 2 of the
                                     string to position 5"

#(a b c x) at: 4 put: #(d e)        "the elements of #(d e)
                                     replace x in the array"

'texas' size between: 2 + 2 and: 4 factorial

set1 add: (i + 1)                   "add new element to set 1"
```

The last two examples above contain messages within messages, while the last example has a message delimited with parentheses. In executing a message

without parentheses, the execution proceeds left to right with unary messages taking precedence followed by binary, and then keyword. Therefore, the messages 'texas' size and 4 factorial are completed before the keyword part between:and: in the last example above.

Methods

Procedures are called methods. They determine the behavior of an object when a message is sent to the object. Methods are the algorithms or sequence of instructions executed by an object. For example, in order to respond to the message 5 + 7, the object 5 must initiate a method to find the sum of the integer numbers 5 and 7. On completion of the operation, the method returns the object 12 to the sending object.

Methods are defined much like procedures in other programming languages using the constructs and syntax of the given OOS. The constructs used to build higher level methods are defined in terms of a number of primitive operations and basic methods provided as part of the OOS. The primitives of an OOS are coded in some host language such as an assembler language or C. For example, the operation for integer addition used in some versions of Smalltalk would be written as

```
+ aNumber
    ^ <SameTypeOfObject self aNumber>
      ifTrue: [<IntegerAddition self aNumber>]
      ifFalse: [super + aNumber]
```

The name of this method is + and the argument is an object of type aNumber. The primitive operation SameTypeOfObject tests whether the two object arguments are of the same type (instances of the same class). The variable self is a temporary variable of an instance of the class it belongs to, Integer. If the two objects are of the same type, the primitive IntegerAddition in the ifTrue block of code is executed and the sum returned. Otherwise, a search for an appropriate method is made by checking the superclass of this class (the class Number) . The up-arrow ^ signifies the quantity to be returned by the method.

A typical OOS may have as many as a few hundred predefined primitives and basic methods combined. We will see examples of some typical methods in the next section.

Classes and Hierarchies

A class is a general object that defines a set of individual (instance) objects which share common characteristics. For example, the class of rabbits contains many individual rabbit objects, each with four legs, long ears, whiskers, and short bushy tails. The class of natural numbers contains many instance objects such as 43,91,2, ... All objects are instances of some class and classes are subclasses of some higher

class, except for a most general root class. The root class for an OOS is the class named Object.

Classes can often be divided into subclasses or merged into superclasses. The class of fruit can be divided into citrus and noncitrus, both of which can be further divided. Fruit is part of the superclass of plant-grown foods which in turn is part of the class of all foods. Classes permit the formation of hierarchies of objects which can be depicted as a tree or taxonomic structure as illustrated in Figure 8.3.

Objects belonging to the same class have the same variables and the same methods. They also respond to the same set of messages called the *protocol* of the class. Each class in a hierarchy inherits the variables and methods of all of its parents or superclasses of the class.

When a message is sent to an object, a check is first made to see if the methods for the object itself or its immediate class can perform the required task. If not, the methods of the nearest superclass are checked. If they are not adequate, the search process continues up the hierarchy recursively until methods have been found or the end of a chain has been reached. If the required methods are not found, an error message is printed.

Some OOSs permit classes to have two or more direct superclasses (Stefik and Bobrow, 1986). For example, StereoSystem may have superclasses of Appliances, LuxuryGoods, and FragileCommodity. As such, a stereo object may inherit characteristics and methods from all three superclasses. When this is the case, an inheritance precedence must be defined among the superclasses. One approach would be to try the leftmost superclass path in the hierarchy first. If applicable methods are not found up this path, the next leftmost path is taken. This process continues progressively shifting to the right until a method is found or failure occurs.

An OOS will have many predefined classes. For example, a few of the classes for the Smalltalk family and their hierarchical structure are depicted in Figure 8.4.

Each of the classes depicted in Figure 8.4 has a number of methods that respond to the protocol for the class. A class may also inherit methods from a superclass. For example, all classes inherit the method "== anObject" which answers true if the receiver and anObject are the same, and answers false otherwise.

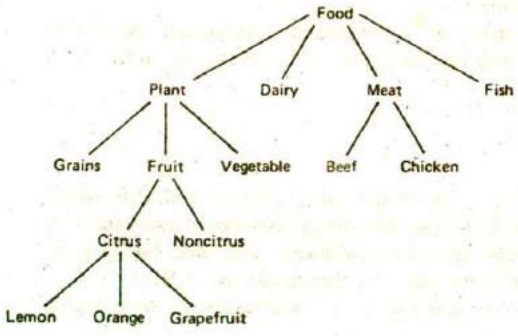


Figure 8.3 A class hierarchy of foods.

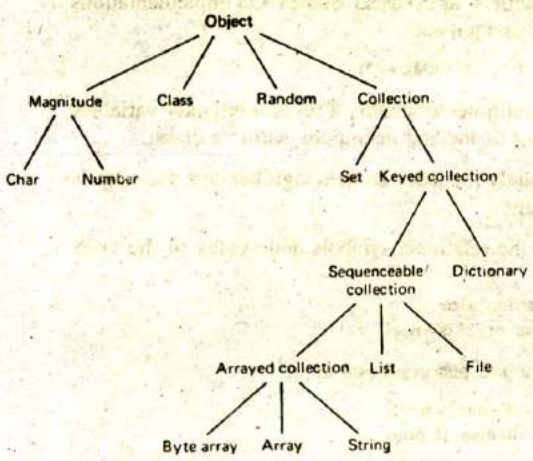


Figure 8.4 Partial hierarchy of predefined OOS classes.

Polymorphism is the capability for different objects to respond to the same message protocols but exhibit their own unique behaviors in response. For example, a message containing the selector moveForward could invoke the forward movement of a ship as well as advancing a piece in a game such as checkers. Both classes use the same message template but respond differently. The only requirement is that the message protocol for the two classes be implemented as required for the given class.

8.4 SIMULATION EXAMPLE USING AN OOS PROGRAM

In this section, we present a simple example of an OOS program to demonstrate some of the features defined above. The example is a program to simulate the seaport operations described in Section 8.2. Before we begin however, we define some additional syntax and operations used in an OOS.

An OOS will have most of the basic programming constructs of a procedural language such as Pascal, including arithmetic and string manipulation operations, assignment and conditional statements, logical expressions, iteration, and so on. A few examples will help to illustrate some of the basic constructs and syntactical conventions.

1. A period is used as a statement separator, not as a terminator.

2. A block object is a sequence of OOS statements (messages) enclosed within square brackets. A block is like an in-line procedure that, when evaluated, returns the last value executed within the block. Blocks are instances of the Block class that execute when sent the message value, value: or value:value:, depending on whether the block has zero, one, or two arguments.

3. Variable assignment is made with := as in Pascal. Some OOS implementations use the back arrow ← like Algol for assignment

```
index := 1      (or index ← 1).
```

4. The vertical bar is used as a delimiter to identify a set of temporary variables used in methods as well as a separator of method definitions within a class.

5. An up arrow which immediately precedes an item signifies that the item is to be returned in response to a message.

6. Boolean relational tests use the relational symbols and syntax of the OOS such as

```
5 < 'string' size
('camel' at: 3) isVowel.
```

7. Conditional statements follow the test condition as in

```
a < b  ifTrue: [a print]
       ifFalse: [b print].
```

8. Typical logical expressions are given by

```
a >= b and: [c <= d]
x isDigit or: [$F <= y and: [y <= $L]]
```

The dollar sign preceding a character identifies character objects.

9. Typical iteration constructs are whileTrue, whileFalse, timesRepeat, and do.

```
i := 1.
[i print. i := i + 2. i <= 10] whileTrue.

1 to: 10 do [[:j] array at: j] print].

paths := 6.
paths timesRepeat [ship move: 100; turn: 360//paths].
```

10. The variable self in a method refers to the receiver object of the message that invokes the method. The variable super is used in a method to invoke a search for a method in an object's superclass.

In addition to the above examples, an OOS will have many special methods for the definition of classes and objects and for the definition of class behaviors and class related tasks.

Simulation of Seaport Operations

An event driven simulation of the seaport operation described in Section 8.2 is a computed sequence of the events of interest which occur at discrete time points.

This system would have as a minimum the three events: (1) cargo ship arrivals, (2) ship berthing operations, and (3) cargo transfer and ship departures. These events are symbolized by the following expressions which will be used in our program.

```
shipArrival
shipDocking
shipDeparture
```

In the interest of clarity, several simplifying assumptions are made for the simulation problem. First, we limit our objects of interest to three classes, namely the class of ships (three types of cargo ships), the group of entities which make up the harbor operations (tugs, docks, cranes, crews, and the like) treated collectively as one object class called HarborOperations, and the class called Simulator. The Simulator class is defined to permit separate simulation runs, that is, separate instances of Simulator.

Second, we assume that ships arriving to find all berths full depart immediately from the system. Ships arriving when at least one of the eight berths is available are scheduled for docking and cargo transfer. Once a ship has been docked, its departure is then scheduled.

To add some realistic randomness to the operation, the time between ship arrivals is assumed to be exponentially distributed with a mean value of 3 time units. The time to dock is assumed to be uniformly distributed with a range of 0.5 to 2.0 time units, and the time to transfer cargo is assumed to be exponentially distributed with a mean of 14 time units. Finally, to simulate three different types of ships, a newly arriving ship is randomly assigned a cargo weight of 10, 20, or 30 thousand tons from an empirical distribution with probabilities 0.2, 0.5, and 0.3, respectively.

The three types of simulated events may occur at any discrete time point, and they may even occur concurrently. To manage these events, we require a system clock to assign scheduled event times and a data structure in which to record all pending events. At any time during the run, the pending event list could include a scheduled ship arrival at current time t_{now} plus some time increment t_1, the berthing of a ship at time $t_{now} + t_2$, and the departures of one or more ships at $t_{now} + t_3$, $t_{now} + t_4$, and so on. Scheduled events are removed from the list of pending events in the order of smallest time value first.

Pending events are held in a dictionary data structure which contains index-value pairs of objects. And, since multiple events may occur at the same time points, we use a set to hold all events indexed by the same time value. Thus, pending events will be stored in a dictionary of indexed sets with each set containing one or more of the basic events.

Messages required to access sets and dictionary objects (collectively referred to as collections) are needed in the program. The messages and the corresponding actions they elicit are as follows.

MESSAGE	RESULTING ACTION
add:	adds an element to the receiver collection, like a list
at:	returns the item in the dictionary whose key matches the argument
at: ifAbsent:	returns the element given by the key in the first argument and evaluates the second argument if no argument exists
at:put:	places the second argument into the receiver collection under the key given by the first argument
first	returns the first element from a Set
includesKey:	returns true if the key is valid for the receiver
isEmpty	returns true if the receiver collection contains no elements
keysDo:	evaluates each key element of the one argument block which follows according to the procedure given in the block
remove:	removes the argument object from the receiver collection
removeKey:	removes the object with the given key from the receiver collection

For output from the simulation, we print the arrival time of each ship, indicating whether it docks or not, each ship departure, and the total cargo transferred at the end of the run.

With the above preliminaries, we now define the three classes and their corresponding methods. We begin with the class Simulator which is the most complicated. To define a class, the word Class is given followed by the class name and (optionally) by the class's immediate superclass; if no superclass is given, the default class Object is assumed. This is followed by a list of local variables within vertical bar delimiters. The method protocol for the class is defined next, with the methods separated by vertical bars. The message template for each method is given as the first item following the vertical bar. When local variables for a method are needed, they follow the message template, also given within vertical bars.

```
Class Simulator
|currentTime eventsPending|
[
  new
    eventsPending := Dictionary new.
    currentTime := 0
|
  time
    ^currentTime
|
  addEvent: event at: eventTime
    (eventsPending includesKey: eventTime)
      ifTrue: [(eventsPending at: eventTime) add: event]
```

```
        ifFalse: [eventsPending at: eventTime
                put: (Set new ; add: event)]
|
  addEvent: event next: delayTime
    self addEvent: event at: currentTime + delayTime
|
  proceed  |minTime eventSet event|
    mintime := 99999.
    eventsPending keysDo:
      [:x| (x < minTime) ifTrue: [minTime := x ]].
    currentTime := minTime.
    eventSet:= eventsPending at:
                    minTime ifAbsent: [' nil].
    event := eventSet first.
    eventSet remove: event.
    (eventSet isEmpty)
        ifTrue: [eventsPending removeKey: minTime].
    self processEvent: event
```

The method responding to the message addEvent checks to see if a time value (key) exists in the dictionary. If so, it adds the new event to the set under the key. If the time does not already exist, a new set is created, and the event is added to the set and the set put in eventsPending. The proceed method finds the smallest event time (key) and retrieves and removes the first element of the set located there. If the resultant set is empty, the key for the empty set is removed. A message is then sent to the processEvent object in the class HarborOperations which is defined next.

```
Class HarborOperation :Simulator
|totalCargo arrivalDistribution dockingDistribution
  serviceDistribution remainingBerths|
[
  new
    totalCargo := 0.
    remainingBerths := 8.
    arrivalDistribution := Exponential new: 3.
    shipDistribution := DiscreteProb new: #(0.2 0.5 0.3)
    dockingDistribution := Uniform new: #(0.5 2.0).
    serviceDistribution := Exponential new: 14.
    self scheduleArrival
|
  scheduleArrival  |newShip time|
    newShip := Ship new.
    self addEvent: [self shipArrival: newShip]
          at: (self time + (arrivalDistribution next))
```

```
    processEvent: event
      event value.
      ('ship arrived at', self time) print.
      totalCargo := totalCargo + (shipSize * 10).
      self scheduleArrival
  |
    reportCargo
      ('total cargo transferred', totalCargo) print
  ]
```

The method new initializes some variables, including the arrival, docking, and service distributions. A sample from a distribution is obtained by sending the distribution the message next. (The programming details for the generation of random samples from all distributions have been omitted in the interest of presenting a more readable example.) The scheduleArrival method sends a message to the Ship class to create a new ship and then adds the event block

```
[self shipArrival: newShip]
```

to the pending event list at the arrival time value. The processEvent method is activated from the proceed method in the Simulator class. It initiates evaluation of the event block stored in the pending list, prints a ship arrival message, and computes the new total for the cargo discharged.

Next, we define the class Ship.

```
Class ship
|shipSize|
[
  new
    shipSize := shipDistribution next
|
  shipSize
    ^shipSize
]
```

With the object classes defined, we can now write the statements for the three object events. The arrival event is initiated from the HarborOperation class. This event then schedules the next operation (docking), which in turn schedules the ship departure event.

```
shipArrival: ship
  (remainingBerths > 0)
    ifTrue: [remainingBerths := remainingBerths - 1.
          self addEvent: [self shipDocking: ship]
           at: (self time + dockingDistribution next)].
    ifFalse: ['all berths occupied, ship departs' print]
```

```
shipDocking: ship
  totalCargo := totalCargo + shipSize.
  self addEvent: [self shipDepart: ship]
        next: (serviceDistribution next)

shipDepart: ship
  'ship departs after cargo transfer' print.
  remainingBerths := remainingBerths + 1
```

To run the simulation program we simply execute the following statements.

```
Simulator new.
port := HarborOperation new
[port time < 720] whileTrue: [port proceed]
```

Note that the message "port proceed" in the whileTrue block is sent to HarborOperation. Since there is no method proceed in this class, it must be inherited from the Simulator class.

An environment for an OOS will usually include all of the basic primitives, class and method definitions, an editor, a browser, window and mouse facilities, and a graphics output.

8.5 OBJECT-ORIENTED LANGUAGES AND SYSTEMS

In addition to the Smalltalk family of OOS languages, a number of other languages have been developed, including object-oriented extensions to LISP dialects and special purpose languages. Typical of the LISP extensions is the FLAVORS add-ons.

OOS with Lisp Extensions

In FLAVORS, classes are created with the defflavor form, and methods of the flavor are created with defmethod. An instance of a flavor is created with a make-instance type of function. For example, to create a new flavor (class) of ships with instance variables *x*-position, *y*-position, *x*-velocity, *y*-velocity, and cargo-capacity, the following expression is evaluated:

```
(defflavor ship (x-position y-position x-velocity
                 y-velocity cargo-capacity))
```

Methods for the ship flavor are written in a similar manner with a defmethod, say for the ship's speed, as

```
(defmethod (ship :speed) ()
(sqrt (+ (* x-velocity x-velocity)
         (* y-velocity y-velocity))))
```

To create an instance of ship one then uses the form

```
(setf ship42 (make-instance 'ship))
```

In addition to user defined methods, three additional predefined methods are available for a flavor. They are used to assign values to instance variables, to initialize variable values, and to get values of the variables. These options are specified in the defflavor statement when the flavor is created. For example, to include these optional methods, the following form is used:

```
(defflavor ship (x-position y-position x-velocity
                 y-velocity cargo-capacity)
    ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)
```

Values for the ship instance variables can now be assigned either with a message or when an instance of the ship is created.

```
(send ship42 :set-cargo-capacity 22.5) or

(setf ship42 (make-instance 'ship :x-position 5.0
                                  :y-position 8.0))
```

Variable assignments can be examined with the describe method, one of the base methods provided with the system.

```
(describe ship42)
#<SHIP 1234567>, an object of flavor SHIP,
has instance variable values:
    X-POSITION          5.0
    Y-POSITION          8.0
    X-VELOCITY          unbound
    Y-VELOCITY          unbound
    CARGO-CAPACITY      22.5
```

Default values can also be assigned to instance variables with the defvar statement.

```
(defvar *default-x-velocity* 12.0)
```

Thus, unless x-velocity is explicitly assigned a value in a make-instance or defflavor statement, it will be given the default value 12.0.

Flavors are defined hierarchically by including one or more superclasses in the second subform of the defflavor statement, thereby permitting flavors to be

"mixed." Inheritance of methods is then achieved much the same as in the Smalltalk case. For example, to create a ship flavor which has two superclass flavors named moving-object and pleasure-craft, the following form would be used:

```
(defflavor ship (x-position y-position passenger-capacity)
                (moving-object pleasure-craft)
  :gettable-instance-variables)
```

If the flavor moving-object has a method for speed, it will be inherited unless a method for speed has been defined explicitly for the ship flavor.

The base flavor for FLAVOR extensions is the vanilla-flavor. This flavor will typically have a number of base methods including a :print-self and describe method as used above. Generic instance variables are also defined for the vanilla-flavor. Through method inheritance and other forms, methods may be combined to give a variety of capabilities for flavors including the execution of some methods just prior to or just after a main method.

Special Purpose OOS Languages

A typical special purpose OOS language is ROSS (for Rand OOS) developed by the Rand Corporation for military battle simulations (Klahr et al, 1980, 1982). This system has been implemented in several dialects of LISP as an interactive simulation system which includes a movie generator and graphics facility. Visual representations can be generated as the simulation is running, and on-the-fly changes can easily be made to a program. It has been used to simulate both air and ground battles.

Messages are sent to objects in ROSS with an "ask" form having the following structure.

```
(ask <object><message>)
```

For example, to send a message to a fighter-base requesting that a fighter be sent to intercept a penetrator, the following message might be sent:

```
(ask fighter-base1 send fighter2 guided by radar3
                   to penetrator2)
```

In response to this message, a method associated with the fighter-base class would be evaluated and appropriate actions would then be initiated through direct computations and embedded message transmissions to other objects.

Object hierarchies which provide variable and method inheritance in much the same way as FLAVORS or Smalltalk can also be created. A simulation program in ROSS would be created in a similar manner to that of the example presented in the previous section, but with a number of basic methods for simulation behaviors

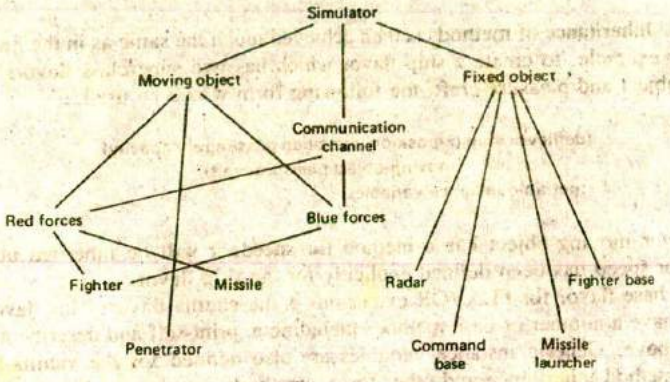


Figure 8.5 An example of a battle class hierarchy.

already predefined. A typical class hierarchy for a battle simulation might be defined as the structure illustrated in Figure 8.5.

8.6 SUMMARY

Object-oriented systems are useful in modeling many real-world situations in which real objects can be put in a one-to-one correspondence with program classes and objects. In an OOS all entities are objects, and objects are protected from other entities. They interact with other objects through an interface which recognizes a set of messages called the protocol for the class. Each class within an OOS will have its own unique behavior which is governed by a set of class methods. Methods and instance variables may be inherited by a class of objects from its parents or superclasses. Multiple inheritance is also supported by many systems.

OOS languages are well suited for certain types of system simulation problems because of the natural way in which OOS programs model real systems. To build a simulation program in an OOS such as Smalltalk, one first defines the object classes required, their hierarchical relationship with each other, and the behaviors of the objects within each class. The events of importance are also defined and the sequence in which they may occur. Message formats for class protocols are then defined, and the specific methods are then coded.

OOS capabilities have been developed for several LISP sytems as add-ons such as found in FLAVORS. Special purpose OOS languages have also been developed such as the ROSS system which was developed to provide capabilities not available in other simulation languages.

EXERCISES

8.1. Show the order of evaluation for the subexpressions given in the following expression:
9/2 between: 8 + 19 sqrt and: 4 * 5

8.2. What values will be printed after the following sequences?

a. *i* := 17
j := [*i* := *i* + 1]
i print

b. *j* value print (after the sequence in a above)

c. *i* value print (after the sequence in b above)

8.3. What is the class of Class? What is the superclass of Class?

8.4. What is the result from typing the following expression?

```
3 + (4 print) ; + 6
```

8.5. A bag is like a set except the same item may occur more than once. One way to implement the class Bag is with a dictionary, where the value contained in the dictionary is the number of times the item occurs in the bag. A partial implementation for a bag is given below. Complete the implementation, keeping in mind that instances of dictionary respond to first and next with values and not keys. The current key is accessible, however, if currentKey is used.

```
Class Bag :Collection
| dict count|
[
    new
      dict:=Dictionary new
|...some methods go here....
|   first
      (count:=dict first) isNil ifTrue:[^ nil].
       count:=count - 1.
       ^ dict currentKey
|   next
      [count notNil] whileTrue:
        [(count>0)
          ifTrue:[count:=count - 1. ^ dict currentKey]
          ifFalse:[count:=dict next]].
        nil
]
```

8.6. One method of defining a discrete probability distribution is to provide the actual sample space elements in a collection. A random sample can then be obtained from the collection entries. Produce a class description for a class called SampleSpace which will be used to randomly select points using the following:

```
sample := SampleSpace new ; define: #(12 9 14 19 11 21)
sample first
12
```

8.7. Modify the simulation program given in Section 8.4 to collect use statistics on the tugs; at the end of the run a printout of average tug usage should be made.