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 ¬, &, 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 V P = P P & P = P
Associativity	(P V Q) V R = P V (Q V R) (P & Q) & R = P & (Q & R)
Commutativity	P V Q = Q V P P & Q = Q & P P ↔ Q = Q ↔ P
Distributivity	P & (Q V R) = (P & Q) V (P & R) P V (Q & R) = (P V Q) & (P V R)
De Morgan's laws	¯(P V Q) = ¯P & ¯Q ¯(P & Q) = ¯P V ¯Q
Conditional elimination	P → Q = ¯P V Q
Bi-conditional elimination	P ↔ Q = (P → Q) & (Q → 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 $\mathbf{S} = \{s_1, \ldots, s_n\}$ we will also say **S** logically implies or logically entails s, written $\mathbf{S} \vdash s$.

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 ↔; 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	¯P	(¯P V Q)	(P → Q)	(Q → P)	(P → Q) & (Q → 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′: ∀x ((E(x) & GE(i(x),1400)) → T(x))
E2′: ∃y (E(y) → S(y))
E3′: ∀xy ((E(x) & P(y)) → ¬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

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

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

{¬MARRIED(x,y) ∨ ¬MOTHER(x,z) ∨ FATHER(y,z),
MARRIED(sue,joe), ¬FATHER(joe,bill)}

where the substitution β = {sue/x, joe/y, bill/z} is used, results in the unit clause ¬MOTHER(sue,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 $P(a_1), P(a_2), \ldots, P(a_k)$, conclude that $\forall x\, P(x)$. More generally, given $P(a_1) \rightarrow Q(b_1), P(a_2) \rightarrow Q(b_2), \ldots, P(a_k) \rightarrow Q(b_k)$, 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 $\neg Q \rightarrow \neg P$