2 THE PREDICATE CALCULUS

We come to the full possession of our power of drawing inferences, the last of our faculties; for it is not so much a natural gift as a long and difficult art.

—C. S. PIERCE

The essential quality of a proof is to compel belief.

—FERMAT

2.0 Introduction

In this chapter we introduce the predicate calculus as a representation language for artificial intelligence. The importance of the predicate calculus was discussed in the introduction to Part II; its advantages include a well-defined *formal semantics* and *sound* and *complete* inference rules. This chapter begins with a brief (optional) review of the propositional calculus (Section 2.1). Section 2.2 defines the syntax and semantics of the predicate calculus. In Section 2.3 we discuss predicate calculus inference rules and their use in problem solving. Finally, the chapter demonstrates the use of the predicate calculus to implement a knowledge base for financial investment advice.

2.1 The Propositional Calculus (optional)

2.1.1 Symbols and Sentences

The propositional calculus and, in the next subsection, the predicate calculus are first of all languages. Using their words, phrases, and sentences, we can represent and reason about properties and relationships in the world. The first step in describing a language is to introduce the pieces that make it up: its set of symbols.

DEFINITION

PROPOSITIONAL CALCULUS SYMBOLS

The *symbols* of propositional calculus are the propositional symbols:

P, Q, R, S, …

truth symbols:

true, false

and connectives:

$\wedge, \vee, \neg, \rightarrow, \equiv$

Propositional symbols denote *propositions*, or statements about the world that may be either true or false, such as "the car is red" or "water is wet." Propositions are denoted by uppercase letters near the end of the English alphabet. Sentences in the propositional calculus are formed from these atomic symbols according to the following rules:

DEFINITION

PROPOSITIONAL CALCULUS SENTENCES

Every propositional symbol and truth symbol is a sentence.

For example: true, P, Q, and R are sentences.

The *negation* of a sentence is a sentence.

For example: $\neg$ P and $\neg$ false are sentences.

The *conjunction*, or *and*, of two sentences is a sentence.

For example: P $\wedge \neg$ P is a sentence.

The *disjunction*, or *or*, of two sentences is a sentence.

For example: P $\vee \neg$ P is a sentence.

The *implication* of one sentence from another is a sentence.

For example: P $\rightarrow$ Q is a sentence.

The *equivalence* of two sentences is a sentence.

For example: P $\vee$ Q $\equiv$ R is a sentence.

Legal sentences are also called *well-formed formulas* or *WFFs*.

In expressions of the form P $\wedge$ Q, P and Q are called the *conjuncts*. In P $\vee$ Q, P and Q are referred to as *disjuncts*. In an implication, P $\rightarrow$ Q, P is the *premise* or *antecedent* and Q, the *conclusion* or *consequent*.

In propositional calculus sentences, the symbols ( ) and [ ] are used to group symbols into subexpressions and so to control their order of evaluation and meaning. For example, $(P \vee Q) \equiv R$ is quite different from $P \vee (Q \equiv R)$, as can be demonstrated using truth tables as we see Section 2.1.2.

An expression is a sentence, or well-formed formula, of the propositional calculus if and only if it can be formed of legal symbols through some sequence of these rules. For example,

$$((P \wedge Q) \rightarrow R) \equiv \neg P \vee \neg Q \vee R$$

is a well-formed sentence in the propositional calculus because:

P, Q, and R are propositions and thus sentences.

$P \wedge Q$, the conjunction of two sentences, is a sentence.

$(P \wedge Q) \rightarrow R$, the implication of a sentence for another, is a sentence.

$\neg P$ and $\neg Q$, the negations of sentences, are sentences.

$\neg P \vee \neg Q$, the disjunction of two sentences, is a sentence.

$\neg P \vee \neg Q \vee R$, the disjunction of two sentences, is a sentence.

$((P \wedge Q) \rightarrow R) \equiv \neg P \vee \neg Q \vee R$, the equivalence of two sentences, is a sentence.

This is our original sentence, which has been constructed through a series of applications of legal rules and is therefore "well formed".

2.1.2 The Semantics of the Propositional Calculus

Section 2.1.1 presented the syntax of the propositional calculus by defining a set of rules for producing legal sentences. In this section we formally define the *semantics* or "meaning" of these sentences. Because AI programs must reason with their representational structures, it is important to demonstrate that the truth of their conclusions depends only on the truth of their initial knowledge or premises, i.e., that logical errors are not introduced by the inference procedures. A precise treatment of semantics is essential to this goal.

A proposition symbol corresponds to a statement about the world. For example, P may denote the statement "it is raining" or Q, the statement "I live in a brown house." A proposition must be either true or false, given some state of the world. The truth value assignment to propositional sentences is called an *interpretation*, an assertion about their truth in some *possible world*.

Formally, an interpretation is a mapping from the propositional symbols into the set $\{T, F\}$. As mentioned in the previous section, the symbols true and false are part of the set of well-formed sentences of the propositional calculus; i.e., they are distinct from the truth value assigned to a sentence. To enforce this distinction, the symbols T and F are used for truth value assignment.

Each possible mapping of truth values onto propositions corresponds to a possible world of interpretation. For example, if P denotes the proposition "it is raining" and Q denotes "I am at work," then the set of propositions {P, Q} has four different functional mappings into the truth values {T, F}. These mappings correspond to four different interpretations. The semantics of propositional calculus, like its syntax, is defined inductively:

DEFINITION

PROPOSITIONAL CALCULUS SEMANTICS

An *interpretation* of a set of propositions is the assignment of a truth value, either T or F, to each propositional symbol.

The symbol true is always assigned T, and the symbol false is assigned F.

The interpretation or truth value for sentences is determined by:

The truth assignment of *negation*, $\neg$ P, where P is any propositional symbol, is F if the assignment to P is T, and T if the assignment to P is F.

The truth assignment of *conjunction*, $\wedge$, is T only when both conjuncts have truth value T; otherwise it is F.

The truth assignment of *disjunction*, $\vee$, is F only when both disjuncts have truth value F; otherwise it is T.

The truth assignment of *implication*, $\rightarrow$, is F only when the premise or symbol before the implication is T and the truth value of the consequent or symbol after the implication is F; otherwise it is T.

The truth assignment of *equivalence*, $\equiv$, is T only when both expressions have the same truth assignment for all possible interpretations; otherwise it is F.

The truth assignments of compound propositions are often described by *truth tables*. A truth table lists all possible truth value assignments to the atomic propositions of an expression and gives the truth value of the expression for each assignment. Thus, a truth table enumerates all possible worlds of interpretation that may be given to an expression. For example, the truth table for P $\wedge$ Q, Figure 2.1, lists truth values for each possible truth assignment of the operands. P $\wedge$ Q is true only when P and Q are both T. Or ($\vee$), not ($\neg$), implies ($\rightarrow$), and equivalence ($\equiv$) are defined in a similar fashion. The construction of these truth tables is left as an exercise.

Two expressions in the propositional calculus are equivalent if they have the same value under all truth value assignments. This equivalence may be demonstrated using truth tables. For example, a proof of the equivalence of P $\rightarrow$ Q and $\neg$ P $\vee$ Q is given by the truth table of Figure 2.2.

By demonstrating that two different sentences in the propositional calculus have identical truth tables, we can prove the following equivalences. For propositional expressions P, Q, and R:

$\neg (\neg P) \equiv P$

$(P \vee Q) \equiv (\neg P \rightarrow Q)$

the contrapositive law: $(P \rightarrow Q) \equiv (\neg Q \rightarrow \neg P)$

de Morgan's law: $\neg (P \vee Q) \equiv (\neg P \wedge \neg Q)$ and $\neg (P \wedge Q) \equiv (\neg P \vee \neg Q)$

the commutative laws: $(P \wedge Q) \equiv (Q \wedge P)$ and $(P \vee Q) \equiv (Q \vee P)$

the associative law: $((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$

the associative law: $((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$

the distributive law: $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$

the distributive law: $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$

Identities such as these can be used to change propositional calculus expressions into a syntactically different but logically equivalent form. These identities may be used instead of truth tables to prove that two expressions are equivalent: find a series of identities that transform one expression into the other. An early AI program, the *Logic Theorist* (Newell and Simon 1956), designed by Newell, Simon, and Shaw, used transformations between equivalent forms of expressions to prove many of the theorems in Whitehead and Russell's *Principia Mathematica* (1950). The ability to change a logical expression into a different form with equivalent truth values is also important when using inference rules (modus ponens, Section 2.3, and resolution, Chapter 14) that require expressions to be in a specific form.

P	Q	P∧Q
T	T	T
T	F	F
F	T	F
F	F	F

Figure 2.1 Truth table for the operator $\wedge$.

P	Q	¬P	¬P∨Q	P⇒Q	(¬P∨Q)=(P⇒Q)
T	T	F	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

Figure 2.2 Truth table demonstrating the equivalence of $P \rightarrow Q$ and $\neg P \vee Q$.

2.2 The Predicate Calculus

In propositional calculus, each atomic symbol (P, Q, etc.) denotes a single proposition. There is no way to access the components of an individual assertion. Predicate calculus provides this ability. For example, instead of letting a single propositional symbol, P, denote the entire sentence "it rained on Tuesday," we can create a predicate weather that describes a relationship between a date and the weather: weather(tuesday, rain). Through inference rules we can manipulate predicate calculus expressions, accessing their individual components and inferring new sentences.

Predicate calculus also allows expressions to contain variables. Variables let us create general assertions about classes of entities. For example, we could state that for all values of X, where X is a day of the week, the statement weather(X, rain) is true; i.e., it rains every day. As we did with the propositional calculus, we will first define the syntax of the language and then discuss its semantics.

2.2.1 The Syntax of Predicates and Sentences

Before defining the syntax of correct expressions in the predicate calculus, we define an alphabet and grammar for creating the *symbols* of the language. This corresponds to the lexical aspect of a programming language definition. Predicate calculus symbols, like the *tokens* in a programming language, are irreducible syntactic elements: they cannot be broken into their component parts by the operations of the language.

In our presentation we represent predicate calculus symbols as strings of letters and digits beginning with a letter. Blanks and nonalphanumeric characters cannot appear within the string, although the underscore, _, may be used to improve readability.

DEFINITION

PREDICATE CALCULUS SYMBOLS

The alphabet that makes up the symbols of the predicate calculus consists of:

1. The set of letters, both upper- and lowercase, of the English alphabet.
2. The set of digits, 0, 1, …, 9.
3. The underscore, _.

Symbols in the predicate calculus begin with a letter and are followed by any sequence of these legal characters.

Legitimate characters in the alphabet of predicate calculus symbols include

a R 6 9 p _ z

Examples of characters not in the alphabet include

% @ / &

Legitimate predicate calculus symbols include

George fire3 tom_and_jerry bill XXXX friends_of

Examples of strings that are not legal symbols are

3jack no blanks allowed ab%cd ***71 duck!!!

Symbols, as we see in Section 2.2.2, are used to denote objects, properties, or relations in a world of discourse. As with most programming languages, the use of "words" that suggest the symbol's intended meaning assists us in understanding program code. Thus, even though l(g,k) and likes(george, kate) are formally equivalent (i.e., they have the same structure), the second can be of great help (for human readers) in indicating what relationship the expression represents. It must be stressed that these descriptive names are intended solely to improve the readability of expressions. The only "meaning" that predicate calculus expressions have is given through their formal semantics.

Parentheses "( )", commas ",", and periods "." are used solely to construct well-formed expressions and do not denote objects or relations in the world. These are called *improper symbols*.

Predicate calculus symbols may represent either *variables, constants, functions*, or *predicates*. Constants name specific objects or properties in the world. Constant symbols must begin with a lowercase letter. Thus george, tree, tall, and blue are examples of well-formed constant symbols. The constants true and false are reserved as *truth symbols*.

Variable symbols are used to designate general classes of objects or properties in the world. Variables are represented by symbols beginning with an uppercase letter. Thus George, BILL, and KAte are legal variables, whereas geORGE and bill are not.

Predicate calculus also allows functions on objects in the world of discourse. Function symbols (like constants) begin with a lowercase letter. Functions denote a mapping of one or more elements in a set (called the *domain* of the function) into a unique element of a second set (the *range* of the function). Elements of the domain and range are objects in the world of discourse. In addition to common arithmetic functions such as addition and multiplication, functions may define mappings between nonnumeric domains.

Note that our definition of predicate calculus symbols does not include numbers or arithmetic operators. The number system is not included in the predicate calculus primitives; instead it is defined axiomatically using "pure" predicate calculus as a basis (Manna and Waldinger 1985). While the particulars of this derivation are of theoretical interest, they are less important to the use of predicate calculus as an AI representation language. For convenience, we assume this derivation and include arithmetic in the language.

Every function symbol has an associated *arity*, indicating the number of elements in the domain mapped onto each element of the range. Thus father could denote a function of arity 1 that maps people onto their (unique) male parent. plus could be a function of arity 2 that maps two numbers onto their arithmetic sum.

A *function expression* is a function symbol followed by its arguments. The arguments are elements from the domain of the function; the number of arguments is equal to the

arity of the function. The arguments are enclosed in parentheses and separated by commas. For example,

```
f(X,Y)
father(david)
price(bananas)
```

are all well-formed function expressions.

Each function expression denotes the mapping of the arguments onto a single object in the range, called the *value* of the function. For example, if father is a unary function, then

```
father(david)
```

is a function expression whose value (in the author's world of discourse) is george. If plus is a function of arity 2, with domain the integers, then

```
plus(2,3)
```

is a function expression whose value is the integer 5. The act of replacing a function with its value is called *evaluation*.

The concept of a predicate calculus symbol or term is formalized in the following definition:

DEFINITION

SYMBOLS and TERMS

Predicate calculus symbols include:

1. *Truth symbols* true and false (these are reserved symbols).
2. *Constant symbols* are symbol expressions having the first character lowercase.
3. *Variable symbols* are symbol expressions beginning with an uppercase character.
4. *Function symbols* are symbol expressions having the first character lowercase. Functions have an attached arity indicating the number of elements of the domain mapped onto each element of the range.

A *function expression* consists of a function constant of arity n, followed by n terms, $t_1, t_2, \cdots, t_n$, enclosed in parentheses and separated by commas.

A predicate calculus *term* is either a constant, variable, or function expression.

Thus, a predicate calculus *term* may be used to denote objects and properties in a problem domain. Examples of terms are:

cat
times(2,3)
X
blue
mother(sarah)
kate

Symbols in predicate calculus may also represent predicates. Predicate symbols, like constants and function names, begin with a lowercase letter. A predicate names a relationship between zero or more objects in the world. The number of objects so related is the arity of the predicate. Examples of predicates are

likes equals on near part_of

An *atomic sentence*, the most primitive unit of the predicate calculus language, is a predicate of arity n followed by n terms enclosed in parentheses and separated by commas. Examples of atomic sentences are

likes(george,kate)
likes(george,susie)
likes(george,sarah,tuesday)
friends(bill,george)
helps(bill,george)
likes(X,george)
likes(X,X)
friends(bill,richard)
friends(father_of(david),father_of(andrew))
helps(richard,bill)

The predicate symbols in these expressions are likes, friends, and helps. A predicate symbol may be used with different numbers of arguments. In this example there are two different likes, one with two and the other with three arguments. When a predicate symbol is used in sentences with different arities, it is considered to represent two different relations. Thus, a predicate relation is defined by its name and its arity. There is no reason that the two different likes cannot make up part of the same description of the world; however, this is usually avoided because it can often cause confusion.

In the predicates above, bill, george, kate, etc., are constant symbols and represent objects in the problem domain. The arguments to a predicate are terms and may also include variables or function expressions. For example,

friends(father_of(david),father_of(andrew))

is a predicate describing a relationship between two objects in a domain of discourse. These arguments are represented as function expressions whose mappings (given that the father_of david is george and the father_of andrew is allen) form the parameters of the predicate. If the function expressions are evaluated, the expression becomes

friends(george,allen)

These ideas are formalized in the following definition.

DEFINITION

PREDICATES and ATOMIC SENTENCES

Predicate symbols are symbols beginning with a lowercase letter.

Predicates have an associated positive integer referred to as the *arity* or "argument number" for the predicate. Predicates with the same name but different arities are considered distinct.

An atomic sentence is a predicate constant of arity n, followed by n terms, $t_1, t_2, \cdots, t_n$, enclosed in parentheses and separated by commas.

The truth values, true and false, are also atomic sentences.

Atomic sentences are also called *atomic expressions*, *atoms*, or *propositions*.

We may combine atomic sentences using logical operators to form *sentences* in the predicate calculus. These are the same logical connectives used in propositional calculus: $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\equiv$.

When a variable appears as an argument in a sentence, it refers to unspecified objects in the domain. First order (Section 2.2.2) predicate calculus includes two symbols, the *variable quantifiers* $\forall$ and $\exists$, that constrain the meaning of a sentence containing a variable. A quantifier is followed by a variable and a sentence, such as

$\exists$ Y friends(Y, peter)
$\forall$ X likes(X, ice_cream)

The *universal quantifier*, $\forall$, indicates that the sentence is true for all values of the variable. In the example, $\forall$ X likes(X, ice_cream) is true for all values in the domain of the definition of X. The *existential quantifier*, $\exists$, indicates that the sentence is true for at least one value in the domain. $\exists$ Y friends(Y, peter) is true if there is at least one object, indicated by Y that is a friend of peter. Quantifiers are discussed in more detail in Section 2.2.2.

Sentences in the predicate calculus are defined inductively.

DEFINITION

PREDICATE CALCULUS SENTENCES

Every atomic sentence is a sentence.

1. If s is a sentence, then so is its negation, $\neg$ s.
2. If s_1 and s_2 are sentences, then so is their conjunction, $s_1 \wedge s_2$.
3. If s_1 and s_2 are sentences, then so is their disjunction, $s_1 \vee s_2$.
4. If s_1 and s_2 are sentences, then so is their implication, $s_1 \rightarrow s_2$.
5. If s_1 and s_2 are sentences, then so is their equivalence, $s_1 \equiv s_2$.

6. If X is a variable and s a sentence, then $\forall$ X s is a sentence.

7. If X is a variable and s a sentence, then $\exists$ X s is a sentence.

Examples of well-formed sentences follow. Let times and plus be function symbols of arity 2 and let equal and foo be predicate symbols with arity 2 and 3, respectively.

plus(two,three) is a function and thus not an atomic sentence.

equal(plus(two,three), five) is an atomic sentence.

equal(plus(2, 3), seven) is an atomic sentence. Note that this sentence, given the standard interpretation of plus and equal, is false. Well-formedness and truth value are independent issues.

$\exists$ X foo(X,two,plus(two,three)) $\wedge$ equal(plus(two,three),five) is a sentence since both conjuncts are sentences.

(foo(two,two,plus(two,three))) $\rightarrow$ (equal(plus(three,two),five) $\equiv$ true) is a sentence because all its components are sentences, appropriately connected by logical operators.

The definition of predicate calculus sentences and the examples just presented suggest a method for verifying that an expression is a sentence. This is written as a recursive algorithm, verify_sentence. verify_sentence takes as argument a candidate expression and returns success if the expression is a sentence.

```
function verify_sentence(expression);
begin
  case
    expression is an atomic sentence: return SUCCESS;
    expression is of the form Q X s, where Q is either ∀ or ∃, X is a variable,
      if verify_sentence(s) returns SUCCESS
      then return SUCCESS
      else return FAIL;
    expression is of the form ¬ s:
      if verify_sentence(s) returns SUCCESS
      then return SUCCESS
      else return FAIL;
    expression is of the form s1 op s2, where op is a binary logical operator:
      if verify_sentence(s1) returns SUCCESS and
        verify_sentence(s2) returns SUCCESS
      then return SUCCESS
      else return FAIL;
    otherwise: return FAIL
  end
end.
```

We conclude this section with an example of the use of predicate calculus to describe a simple world. The domain of discourse is a set of family relationships in a biblical genealogy:

mother(eve,abel)
mother(eve,cain)
father(adam,abel)
father(adam,cain)

$\forall$ X $\forall$ Y father(X, Y) $\vee$ mother(X, Y) $\rightarrow$ parent(X, Y)
$\forall$ X $\forall$ Y $\forall$ Z parent(X, Y) $\wedge$ parent(X, Z) $\rightarrow$ sibling(Y, Z)

In this example we use the predicates mother and father to define a set of parent–child relationships. The implications give general definitions of other relationships, such as parent and sibling, in terms of these predicates. Intuitively, it is clear that these implications can be used to infer facts such as sibling(cain,abel). To formalize this process so that it can be performed on a computer, care must be taken to define inference algorithms and to ensure that such algorithms indeed draw correct conclusions from a set of predicate calculus assertions. In order to do so, we define the semantics of the predicate calculus (Section 2.2.2) and then address the issue of inference rules (Section 2.3).

2.2.2 A Semantics for the Predicate Calculus

Having defined well-formed expressions in the predicate calculus, it is important to determine their meaning in terms of objects, properties, and relations in the world. Predicate calculus semantics provide a formal basis for determining the truth value of well-formed expressions. The truth of expressions depends on the mapping of constants, variables, predicates, and functions into objects and relations in the domain of discourse. The truth of relationships in the domain determines the truth of the corresponding expressions.

For example, information about a person, George, and his friends Kate and Susie may be expressed by

friends(george,susie)
friends(george,kate)

If it is indeed true that George is a friend of Susie and George is a friend of Kate then these expressions would each have the truth value (assignment) T. If George is a friend of Susie but not of Kate, then the first expression would have truth value T and the second would have truth value F.

To use the predicate calculus as a representation for problem solving, we describe objects and relations in the domain of interpretation with a set of well-formed expressions. The terms and predicates of these expressions denote objects and relations in the domain. This database of predicate calculus expressions, each having truth value T, describes the

"state of the world". The description of George and his friends is a simple example of such a database. Another example is the *blocks world* in the introduction to Part II.

Based on these intuitions, we formally define the semantics of predicate calculus. First, we define an *interpretation* over a domain D. Then we use this interpretation to determine the *truth value assignment* of sentences in the language.

DEFINITION

INTERPRETATION

Let the domain D be a nonempty set.

An *interpretation* over D is an assignment of the entities of D to each of the constant, variable, predicate, and function symbols of a predicate calculus expression, such that:

1. Each constant is assigned an element of D.
2. Each variable is assigned to a nonempty subset of D; these are the allowable substitutions for that variable.
3. Each function f of arity m is defined on m arguments of D and defines a mapping from D^m into D.
4. Each predicate p of arity n is defined on n arguments from D and defines a mapping from D^n into {T, F}.

Given an interpretation, the meaning of an expression is a truth value assignment over the interpretation.

DEFINITION

TRUTH VALUE OF PREDICATE CALCULUS EXPRESSIONS

Assume an expression E and an interpretation I for E over a nonempty domain D. The truth value for E is determined by:

1. The value of a constant is the element of D it is assigned to by I.
2. The value of a variable is the set of elements of D it is assigned to by I.
3. The value of a function expression is that element of D obtained by evaluating the function for the parameter values assigned by the interpretation.
4. The value of truth symbol "true" is T and "false" is F.
5. The value of an atomic sentence is either T or F, as determined by the interpretation I.

6. The value of the negation of a sentence is T if the value of the sentence is F and is F if the value of the sentence is T.

7. The value of the conjunction of two sentences is T if the value of both sentences is T and is F otherwise.

8.–10. The truth value of expressions using $\vee$, $\rightarrow$, and $\equiv$ is determined from the value of their operands as defined in Section 2.1.2.

Finally, for a variable X and a sentence S containing X:

11. The value of $\forall$ X S is T if S is T for all assignments to X under I, and it is F otherwise.

12. The value of $\exists$ X S is T if there is an assignment to X in the interpretation under which S is T; otherwise it is F.

Quantification of variables is an important part of predicate calculus semantics. When a variable appears in a sentence, such as X in likes(george,X), the variable functions as a placeholder. Any constant allowed under the interpretation can be substituted for it in the expression. Substituting kate or susie for X in likes(george,X) forms the statements likes(george,kate) and likes(george,susie) as we saw earlier.

The variable X stands for all constants that might appear as the second parameter of the sentence. This variable name might be replaced by any other variable name, such as Y or PEOPLE, without changing the meaning of the expression. Thus the variable is said to be a *dummy*. In the predicate calculus, variables must be *quantified* in either of two ways: *universally* or *existentially*. A variable is considered *free* if it is not within the scope of either the universal or existential quantifiers. An expression is *closed* if all of its variables are quantified. A *ground expression* has no variables at all. In the predicate calculus all variables must be quantified.

Parentheses are often used to indicate the *scope* of quantification, that is, the instances of a variable name over which a quantification holds. Thus, for the symbol indicating universal quantification, $\forall$:

$$\forall X (p(X) \vee q(Y) \rightarrow r(X))$$

indicates that X is universally quantified in both p(X) and r(X).

Universal quantification introduces problems in computing the truth value of a sentence, because all the possible values of a variable symbol must be tested to see whether the expression remains true. For example, to test the truth value of $\forall$ X likes(george,X), where X ranges over the set of all humans, all possible values for X must be tested. If the domain of an interpretation is infinite, exhaustive testing of all substitutions to a universally quantified variable is computationally impossible: the algorithm may never halt. Because of this problem, the predicate calculus is said to be *undecidable*. Because the propositional calculus does not support variables, sentences can only have a finite number of truth assignments, and we can exhaustively test all these possible assignments. This is done with the truth table, Section 2.1.

As seen in Section 2.2.1 variables may also be quantified *existentially,* indicated by the symbol ∃. In the existential case the expression containing the variable is said to be true for at least one substitution from its domain of definition. The scope of an existentially quantified variable is also indicated by enclosing the quantified occurrences of the variable in parentheses.

Evaluating the truth of an expression containing an existentially quantified variable may be no easier than evaluating the truth of expressions containing universally quantified variables. Suppose we attempt to determine the truth of the expression by trying substitutions until one is found that makes the expression true. If the domain of the variable is infinite and the expression is false under all substitutions, the algorithm will never halt.

Several relationships between negation and the universal and existential quantifiers are given below. These relationships are used in resolution refutation systems described in Chapter 14. The notion of a variable name as a dummy symbol that stands for a set of constants is also noted. For predicates p and q and variables X and Y:

$$\neg \exists X\ p(X) \equiv \forall X \neg p(X)$$

$$\neg \forall X\ p(X) \equiv \exists X \neg p(X)$$

$$\exists X\ p(X) \equiv \exists Y\ p(Y)$$

$$\forall X\ q(X) \equiv \forall Y\ q(Y)$$

$$\forall X\ (p(X) \wedge q(X)) \equiv \forall X\ p(X) \wedge \forall Y\ q(Y)$$

$$\exists X\ (p(X) \vee q(X)) \equiv \exists X\ p(X) \vee \exists Y\ q(Y)$$

In the language we have defined, universally and existentially quantified variables may refer only to objects (constants) in the domain of discourse. Predicate and function names may not be replaced by quantified variables. This language is called the *first-order predicate calculus*.

DEFINITION

FIRST-ORDER PREDICATE CALCULUS

First-order predicate calculus allows quantified variables to refer to objects in the domain of discourse and not to predicates or functions.

For example,

$$\forall (\text{Likes})\ \text{Likes}(\text{george},\text{kate})$$

is not a well-formed expression in the first-order predicate calculus. There are *higher-order* predicate calculi where such expressions are meaningful. Some researchers

(McCarthy 1968, Appelt 1985) have used higher-order languages to represent knowledge in natural language understanding programs.

Many grammatically correct English sentences can be represented in the first-order predicate calculus using the symbols, connectives, and variable symbols defined in this section. It is important to note that there is no unique mapping of sentences into predicate calculus expressions; in fact, an English sentence may have any number of different predicate calculus representations. A major challenge for AI programmers is to find a scheme for using these predicates that optimizes the expressiveness and efficiency of the resulting representation. Examples of English sentences represented in predicate calculus are:

If it doesn't rain on Monday, Tom will go to the mountains.
$\neg$ weather(rain, monday) $\rightarrow$ go(tom, mountains)

Emma is a Doberman pinscher and a good dog.
gooddog(emma) $\wedge$ isa(emma, doberman)

All basketball players are tall.
$\forall$ X (basketball_player(X) $\rightarrow$ tall(X))

Some people like anchovies.
$\exists$ X (person(X) $\wedge$ likes(X, anchovies))

If wishes were horses, beggars would ride.
equal(wishes, horses) $\rightarrow$ ride(beggars)

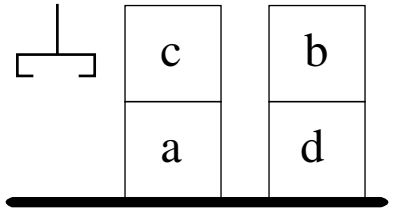
Nobody likes taxes.
$\neg \exists$ X likes(X, taxes)

2.2.3 A "Blocks World" Example of Semantic Meaning

We conclude this section by giving an extended example of a truth value assignment to a set of predicate calculus expressions. Suppose we want to model the blocks world of Figure 2.3 to design, for example, a control algorithm for a robot arm. We can use predicate calculus sentences to represent the qualitative relationships in the world: does a given block have a clear top surface? can we pick up block a? etc. Assume that the computer has knowledge of the location of each block and the arm and is able to keep track of these locations (using three-dimensional coordinates) as the hand moves blocks about the table.

We must be very precise about what we are proposing with this "blocks world" example. First, we are creating a set of predicate calculus expressions that is to represent a static snapshot of the blocks world problem domain. As we will see in Section 2.3, this set of blocks offers an *interpretation* and a possible *model* for the set of predicate calculus expressions.

Second, the predicate calculus is *declarative*, that is, there is no assumed timing or order for considering each expression. Nonetheless, in the planning section of this book, Section 8.4, we will add a "procedural semantics", or a clearly specified methodology for evaluating these expressions over time. A concrete example of a procedural semantics for predicate calculus expressions is Prolog, Section 14.3. This situation calculus we are



on(c,a)
on(b,d)
ontable(a)
ontable(d)
clear(b)
clear(c)
hand_empty

Figure 2.3 A blocks world with its predicate calculus description.

creating will introduce a number of issues, including the *frame problem* and the issue of *non-monotonicity* of logic interpretations, that will be addressed later in this book. For this example, however, it is sufficient to say that our predicate calculus expressions will be evaluated in a top-down and left-to-right fashion.

To pick up a block and stack it on another block, both blocks must be clear. In Figure 2.3, block a is not clear. Because the arm can move blocks, it can change the state of the world and clear a block. Suppose it removes block c from block a and updates the knowledge base to reflect this by deleting the assertion on(c,a). The program needs to be able to infer that block a has become clear.

The following rule describes when a block is clear:

$$\forall X (\neg \exists Y \text{ on}(Y,X) \rightarrow \text{clear}(X))$$

That is, for all X, X is clear if there does not exist a Y such that Y is on X.

This rule not only defines what it means for a block to be clear but also provides a basis for determining how to clear blocks that are not. For example, block d is not clear, because if variable X is given value d, substituting b for Y will make the statement false. Therefore, to make this definition true, block b must be removed from block d. This is easily done because the computer has a record of all the blocks and their locations.

Besides using implications to define when a block is clear, other rules may be added that describe operations such as stacking one block on top of another. For example: to stack X on Y, first empty the hand, then clear X, then clear Y, and then pick_up X and put_down X on Y.

$$\forall X \forall Y ((\text{hand\_empty} \wedge \text{clear}(X) \wedge \text{clear}(Y) \wedge \text{pick\_up}(X) \wedge \text{put\_down}(X,Y)) \rightarrow \text{stack}(X,Y))$$

Note that in implementing the above description it is necessary to "attach" an action of the robot arm to each predicate such as pick_up(X). As noted previously, for such an implementation it was necessary to augment the semantics of predicate calculus by requiring that the actions be performed in the order in which they appear in a rule premise. However, much is gained by separating these issues from the use of predicate calculus to define the relationships and operations in the domain.

Figure 2.3 gives a semantic interpretation of these predicate calculus expressions. This interpretation maps the constants and predicates in the set of expressions into a domain D, here the blocks and relations between them. The interpretation gives truth value T to each expression in the description. Another interpretation could be offered by a different set of blocks in another location, or perhaps by a team of four acrobats. The important question is not the uniqueness of interpretations, but whether the interpretation provides a truth value for all expressions in the set and whether the expressions describe the world in sufficient detail that all necessary inferences may be carried out by manipulating the symbolic expressions. The next section uses these ideas to provide a formal basis for predicate calculus inference rules.

2.3 Using Inference Rules to Produce Predicate Calculus Expressions

2.3.1 Inference Rules

The semantics of the predicate calculus provides a basis for a formal theory of *logical inference*. The ability to infer new correct expressions from a set of true assertions is an important feature of the predicate calculus. These new expressions are correct in that they are *consistent* with all previous interpretations of the original set of expressions. First we discuss these ideas informally and then we create a set of definitions to make them precise.

An interpretation that makes a sentence true is said to *satisfy* that sentence. An interpretation that satisfies every member of a set of expressions is said to satisfy the set. An expression X *logically follows* from a set of predicate calculus expressions S if every interpretation that satisfies S also satisfies X. This notion gives us a basis for verifying the correctness of rules of inference: the function of logical inference is to produce new sentences that logically follow a given set of predicate calculus sentences.

It is important that the precise meaning of *logically follows* be understood: for expression X to *logically follow* S, it must be true for every interpretation that satisfies the original set of expressions S. This would mean, for example, that any new predicate calculus expression added to the blocks world of Figure 2.3 must be true in that world as well as in any other interpretation that that set of expressions may have.

The term itself, "logically follows," may be a bit confusing. It does not mean that X is deduced from or even that it is deducible from S. It simply means that X is true for every

interpretation that satisfies S. However, because systems of predicates can have a potentially infinite number of possible interpretations, it is seldom practical to try all interpretations. Instead, *inference rules* provide a computationally feasible way to determine when an expression, a component of an interpretation, logically follows for that interpretation. The concept "logically follows" provides a formal basis for proofs of the soundness and correctness of inference rules.

An inference rule is essentially a mechanical means of producing new predicate calculus sentences from other sentences. That is, inference rules produce new sentences based on the syntactic form of given logical assertions. When every sentence X produced by an inference rule operating on a set S of logical expressions logically follows from S, the inference rule is said to be *sound*.

If the inference rule is able to produce every expression that logically follows from S, then it is said to be *complete*. *Modus ponens*, to be introduced below, and *resolution*, introduced in Chapter 11, are examples of inference rules that are sound and, when used with appropriate application strategies, complete. Logical inference systems generally use sound rules of inference, although later chapters (4, 9, and 15) examine heuristic reasoning and commonsense reasoning, both of which relax this requirement.

We formalize these ideas through the following definitions.

DEFINITION

SATISFY, MODEL, VALID, INCONSISTENT

For a predicate calculus expression X and an interpretation I:

If X has a value of T under I and a particular variable assignment, then I is said to *satisfy* X.

If I satisfies X for all variable assignments, then I is a *model* of X.

X is *satisfiable* if and only if there exist an interpretation and variable assignment that satisfy it; otherwise, it is *unsatisfiable*.

A set of expressions is *satisfiable* if and only if there exist an interpretation and variable assignment that satisfy every element.

If a set of expressions is not satisfiable, it is said to be *inconsistent*.

If X has a value T for all possible interpretations, X is said to be *valid*.

In the blocks world example of Figure 2.3, the blocks world was a model for its logical description. All of the sentences in the example were true under this interpretation. When a knowledge base is implemented as a set of true assertions about a problem domain, that domain is a model for the knowledge base.

The expression $\exists X (p(X) \wedge \neg p(X))$ is inconsistent, because it cannot be satisfied under any interpretation or variable assignment. On the other hand, the expression $\forall X (p(X) \vee \neg p(X))$ is valid.

The truth table method can be used to test validity for any expression not containing variables. Because it is not always possible to decide the validity of expressions containing variables (as mentioned above, the process may not terminate), the full predicate calculus is "undecidable." There are *proof procedures*, however, that can produce any expression that logically follows from a set of expressions. These are called *complete* proof procedures.

DEFINITION

PROOF PROCEDURE

A *proof procedure* is a combination of an inference rule and an algorithm for applying that rule to a set of logical expressions to generate new sentences.

We present proof procedures for the *resolution* inference rule in Chapter 11.

Using these definitions, we may formally define "logically follows."

DEFINITION

LOGICALLY FOLLOWS, SOUND, and COMPLETE

A predicate calculus expression X *logically follows* from a set S of predicate calculus expressions if every interpretation and variable assignment that satisfies S also satisfies X.

An inference rule is *sound* if every predicate calculus expression produced by the rule from a set S of predicate calculus expressions also logically follows from S.

An inference rule is *complete* if, given a set S of predicate calculus expressions, the rule can infer every expression that logically follows from S.

Modus ponens is a sound inference rule. If we are given an expression of the form $P \rightarrow Q$ and another expression of the form P such that both are true under an interpretation I, then modus ponens allows us to infer that Q is also true for that interpretation. Indeed, because modus ponens is sound, Q is true for *all* interpretations for which P and $P \rightarrow Q$ are true.

Modus ponens and a number of other useful inference rules are defined below.

DEFINITION

MODUS PONENS, MODUS TOLLENS, AND ELIMINATION, AND INTRODUCTION, and UNIVERSAL INSTANTIATION

If the sentences P and $P \rightarrow Q$ are known to be true, then *modus ponens* lets us infer Q.

Under the inference rule *modus tollens*, if $P \rightarrow Q$ is known to be true and Q is known to be false, we can infer that P is false: $\neg P$.

And elimination allows us to infer the truth of either of the conjuncts from the truth of a conjunctive sentence. For instance, $P \wedge Q$ lets us conclude P and Q are true.

And introduction lets us infer the truth of a conjunction from the truth of its conjuncts. For instance, if P and Q are true, then $P \wedge Q$ is true.

Universal instantiation states that if any universally quantified variable in a true sentence, say $p(X)$, is replaced by an appropriate term from the domain, the result is a true sentence. Thus, if a is from the domain of X, $\forall X\ p(X)$ lets us infer $p(a)$.

As a simple example of the use of modus ponens in the propositional calculus, assume the following observations: "if it is raining then the ground is wet" and "it is raining." If P denotes "it is raining" and Q is "the ground is wet" then the first expression becomes $P \rightarrow Q$. Because it is indeed now raining (P is true), our set of axioms becomes

$P \rightarrow Q$
P

Through an application of modus ponens, the fact that the ground is wet (Q) may be added to the set of true expressions.

Modus ponens can also be applied to expressions containing variables. Consider as an example the common syllogism "all men are mortal and Socrates is a man; therefore Socrates is mortal." "All men are mortal" may be represented in predicate calculus by:

$\forall X\ (man(X) \rightarrow mortal(X))$.

"Socrates is a man" is

$man(socrates)$.

Because the X in the implication is universally quantified, we may substitute any value in the domain for X and still have a true statement under the inference rule of universal instantiation. By substituting $socrates$ for X in the implication, we infer the expression

$man(socrates) \rightarrow mortal(socrates)$.

We can now apply modus ponens and infer the conclusion $mortal(socrates)$. This is added to the set of expressions that logically follow from the original assertions. An algorithm called *unification* can be used by an automated problem solver to determine that $socrates$ may be substituted for X in order to apply modus ponens. Unification is discussed in Section 2.3.2.

Chapter 14 discusses a more powerful rule of inference called *resolution*, which is the basis of many automated reasoning systems.

2.3.2 Unification

To apply inference rules such as modus ponens, an inference system must be able to determine when two expressions are the same or *match*. In propositional calculus, this is trivial: two expressions match if and only if they are syntactically identical. In predicate calculus, the process of matching two sentences is complicated by the existence of variables in the expressions. Universal instantiation allows universally quantified variables to be replaced by terms from the domain. This requires a decision process for determining the variable substitutions under which two or more expressions can be made identical (usually for the purpose of applying inference rules).

Unification is an algorithm for determining the substitutions needed to make two predicate calculus expressions match. We have already seen this done in the previous subsection, where socrates in man(socrates) was substituted for X in ∀ X(man(X) ⇒ mortal(X)). This allowed the application of modus ponens and the conclusion mortal(socrates). Another example of unification was seen previously when dummy variables were discussed. Because p(X) and p(Y) are equivalent, Y may be substituted for X to make the sentences match.

Unification and inference rules such as modus ponens allow us to make inferences on a set of logical assertions. To do this, the logical database must be expressed in an appropriate form.

An essential aspect of this form is the requirement that all variables be universally quantified. This allows full freedom in computing substitutions. Existentially quantified variables may be eliminated from sentences in the database by replacing them with the constants that make the sentence true. For example, ∃ X parent(X,tom) could be replaced by the expression parent(bob,tom) or parent(mary,tom), assuming that bob and mary are tom s parents under the interpretation.

The process of eliminating existentially quantified variables is complicated by the fact that the value of these substitutions may depend on the value of other variables in the expression. For example, in the expression ∀ X ∃ Y mother(X,Y), the value of the existentially quantified variable Y depends on the value of X. *Skolemization* replaces each existentially quantified variable with a function that returns the appropriate constant as a function of some or all of the other variables in the sentence. In the above example, because the value of Y depends on X, Y could be replaced by a *skolem function*, f, of X. This yields the predicate ∀ X mother(X,f(X)). Skolemization, a process that can also bind universally quantified variables to constants, is discussed in more detail in Section 14.2.

Once the existentially quantified variables have been removed from a logical database, unification may be used to match sentences in order to apply inference rules such as modus ponens.

Unification is complicated by the fact that a variable may be replaced by any term, including other variables and function expressions of arbitrary complexity. These expressions may themselves contain variables. For example, father(jack) may be substituted for X in man(X) to infer that jack s father is mortal.

Some instances of the expression

```
foo(X,a,goo(Y)).
```

generated by legal substitutions are given below:

1) foo(fred,a,goo(Z))
2) foo(W,a,goo(jack))
3) foo(Z,a,goo(moo(Z)))

In this example, the substitution instances or *unifications* that would make the initial expression identical to each of the other three are written as the sets:

1) {fred/X, Z/Y}
2) {W/X, jack/Y}
3) {Z/X, moo(Z)/Y}

The notation X/Y,… indicates that X is substituted for the variable Y in the original expression. Substitutions are also referred to as *bindings*. A variable is said to be *bound* to the value substituted for it.

In defining the unification algorithm that computes the substitutions required to match two expressions, a number of issues must be taken into account. First, although a constant may be systematically substituted for a variable, any constant is considered a "ground instance" and may not be replaced. Neither can two different ground instances be substituted for one variable. Second, a variable cannot be unified with a term containing that variable. X cannot be replaced by p(X) as this creates an infinite expression: p(p(p(p(...X)...))). The test for this situation is called the *occurs check*.

Furthermore, a problem-solving process often requires multiple inferences and, consequently, multiple successive unifications. Logic problem solvers must maintain consistency of variable substitutions. It is important that any unifying substitution be made consistently across all occurrences within the scope of the variable in both expressions being matched. This was seen before when socrates was substituted not only for the variable X in man(X) but also for the variable X in mortal(X).

Once a variable has been bound, future unifications and inferences must take the value of this binding into account. If a variable is bound to a constant, that variable may not be given a new binding in a future unification. If a variable X_1 is substituted for another variable X_2 and at a later time X_1 is replaced by a constant, then X_2 must also reflect this binding. The set of substitutions used in a sequence of inferences is important, because it may contain the answer to the original query (Section 14.2.5). For example, if p(a,X) unifies with the premise of p(Y,Z) ⇒ q(Y,Z) with substitution {a/Y, X/Z}, modus ponens lets us infer q(a,X) under the same substitution. If we match this result with the premise of q(W,b) ⇒ r(W,b), we infer r(a,b) under the substitution set {a/W, b/X}.

Another important concept is the *composition* of unification substitutions. If S and S′ are two substitution sets, then the composition of S and S′ (written SS′) is obtained by applying S′ to the elements of S and adding the result to S. Consider the example of composing the following three sets of substitutions:

{X/Y, W/Z}, {V/X}, {a/V, f(b)/W}.

Composing the third set, {a/V, f(b)/W}, with the second, {V/X}, produces:

{a/X, a/V, f(b)/W}.

Composing this result with the first set, {X/Y, W/Z}, produces the set of substitutions:

{a/Y, a/X, a/V, f(b)/Z, f(b)/W}.

Composition is the method by which unification substitutions are combined and returned in the recursive function unify, presented next. Composition is associative but not commutative. The exercises present these issues in more detail.

A further requirement of the unification algorithm is that the unifier be as general as possible: that the *most general unifier* be found. This is important, as will be seen in the next example, because, if generality is lost in the solution process, it may lessen the scope of the eventual solution or even eliminate the possibility of a solution entirely.

For example, in unifying p(X) and p(Y) any constant expression such as {fred/X, fred/Y} will work. However, fred is not the most general unifier; any variable would produce a more general expression: {Z/X, Z/Y}. The solutions obtained from the first substitution instance would always be restricted by having the constant fred limit the resulting inferences; i.e., fred would be a unifier, but it would lessen the generality of the result.

DEFINITION

MOST GENERAL UNIFIER (mgu)

If s is any unifier of expressions E, and g is the most general unifier of that set of expressions, then for s applied to E there exists another unifier s′ such that Es = Egs′, where Es and Egs′ are the composition of unifiers applied to the expression E.

The most general unifier for a set of expressions is unique except for alphabetic variations; i.e., whether a variable is eventually called X or Y really does not make any difference to the generality of the resulting unifications.

Unification is important for any artificial intelligence problem solver that uses the predicate calculus for representation. Unification specifies conditions under which two (or more) predicate calculus expressions may be said to be equivalent. This allows use of inference rules, such as *resolution*, with logic representations, a process that often requires backtracking to find all possible interpretations.

We next present pseudo-code for a function, unify, that can compute the unifying substitutions (when this is possible) between two predicate calculus expressions. Unify takes as arguments two expressions in the predicate calculus and returns either the most general set of unifying substitutions or the constant FAIL if no unification is possible. It is defined as a recursive function: first, it recursively attempts to unify the initial components of the expressions. If this succeeds, any substitutions returned by this unification are applied to the remainder of both expressions. These are then passed in a second recursive call to

unify, which attempts to complete the unification. The recursion stops when either argument is a symbol (a predicate, function name, constant, or variable) or the elements of the expression have all been matched.

To simplify the manipulation of expressions, the algorithm assumes a slightly modified syntax. Because unify simply performs syntactic pattern matching, it can effectively ignore the predicate calculus distinction between predicates, functions, and arguments. By representing an expression as a *list* (an ordered sequence of elements) with the predicate or function name as the first element followed by its arguments, we simplify the manipulation of expressions. Expressions in which an argument is itself a predicate or function expression are represented as lists within the list, thus preserving the structure of the expression. Lists are delimited by parentheses, ( ), and list elements are separated by spaces. Examples of expressions in both predicate calculus, PC, and list syntax include:

PC SYNTAX	LIST SYNTAX
p(a,b)	(p a b)
p(f(a),g(X,Y))	(p (f a) (g X Y))
equal(eve,mother(cain))	(equal eve (mother cain))

We next present the function unify:

```
function unify(E1, E2);
  begin
    case
      both E1 and E2 are constants or the empty list:                %recursion stops
        if E1 = E2 then return {}
          else return FAIL;
      E1 is a variable:
        if E1 occurs in E2 then return FAIL
          else return {E2/E1};
      E2 is a variable:
        if E2 occurs in E1 then return FAIL
          else return {E1/E2}
      either E1 or E2 are empty then return FAIL      %the lists are of different sizes
      otherwise:                                            %both E1 and E2 are lists
        begin
            HE1 := first element of E1;
            HE2 := first element of E2;
            SUBS1 := unify(HE1,HE2);
            if SUBS1 : = FAIL then return FAIL;
            TE1 := apply(SUBS1, rest of E1);
            TE2 : = apply (SUBS1, rest of E2);
            SUBS2 : = unify(TE1, TE2);
            if SUBS2 = FAIL then return FAIL;
                else return composition(SUBS1,SUBS2)
        end
    end                                                                  %end case
  end
```

2.3.3 A Unification Example

The behavior of the preceding algorithm may be clarified by tracing the call

```
unify((parents X (father X) (mother bill)), (parents bill (father bill) Y)).
```

When unify is first called, because neither argument is an atomic symbol, the function will attempt to recursively unify the first elements of each expression, calling

```
unify(parents, parents).
```

This unification succeeds, returning the empty substitution, { }. Applying this to the remainder of the expressions creates no change; the algorithm then calls

```
unify((X (father X) (mother bill)), (bill (father bill) Y)).
```

A tree depiction of the execution at this stage appears in Figure 2.4.

In the second call to unify, neither expression is atomic, so the algorithm separates each expression into its first component and the remainder of the expression. This leads to the call

```
unify(X, bill).
```

This call succeeds, because both expressions are atomic and one of them is a variable. The call returns the substitution {bill/X}. This substitution is applied to the remainder of each expression and unify is called on the results, as in Figure 2.5:

```
unify(((father bill) (mother bill)), ((father bill)Y)).
```

The result of this call is to unify (father bill) with (father bill). This leads to the calls

```
unify(father, father)
unify(bill, bill)
unify(( ), ( ))
```

All of these succeed, returning the empty set of substitutions as seen in Figure 2.6.

Unify is then called on the remainder of the expressions:

```
unify(((mother bill)), (Y)).
```

This, in turn, leads to calls

```
unify((mother bill), Y)
```

```
unify(( ),( )).
```

In the first of these, (mother bill) unifies with Y. Notice that unification substitutes the whole *structure* (mother bill) for the variable Y. Thus, unification succeeds and returns the substitution {(mother bill)/Y}. The call

```
unify(( ),( ))
```

returns { }. All the substitutions are composed as each recursive call terminates, to return the answer {bill/X (mother bill)/Y}. A trace of the entire execution appears in Figure 2.6. Each call is numbered to indicate the order in which it was made; the substitutions returned by each call are noted on the arcs of the tree.

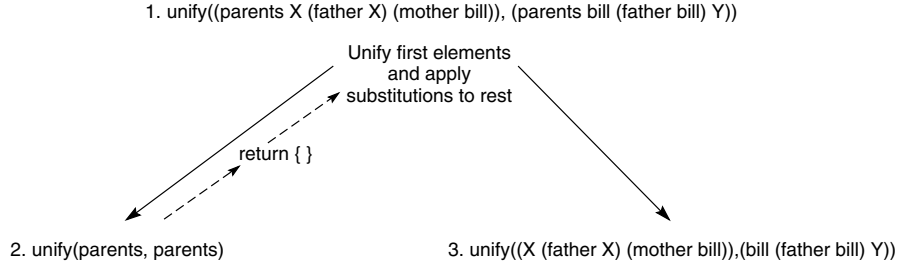


Figure 2.4 Initial steps in the unification of (parents X (father X) (mother bill)) and (parents bill (father bill) Y).

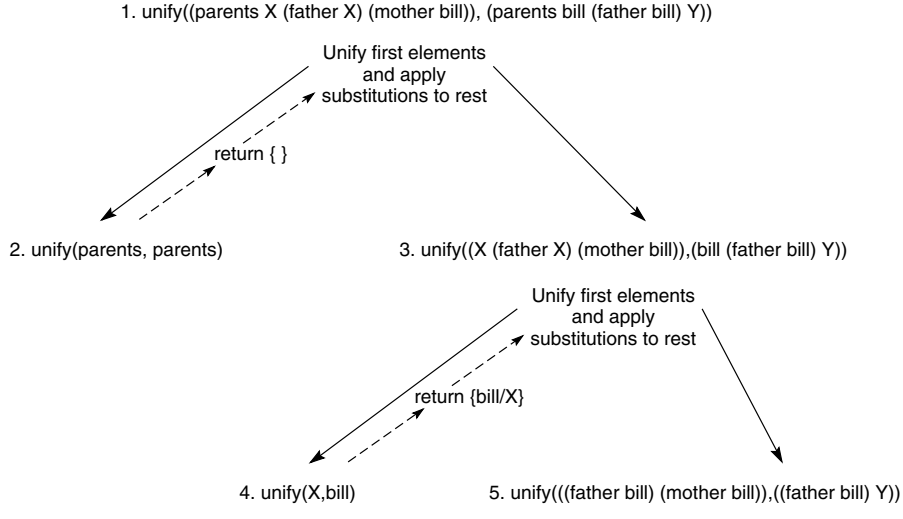


Figure 2.5 Further steps in the unification of (parents X (father X) (mother bill)) and (parents bill (father bill) Y).

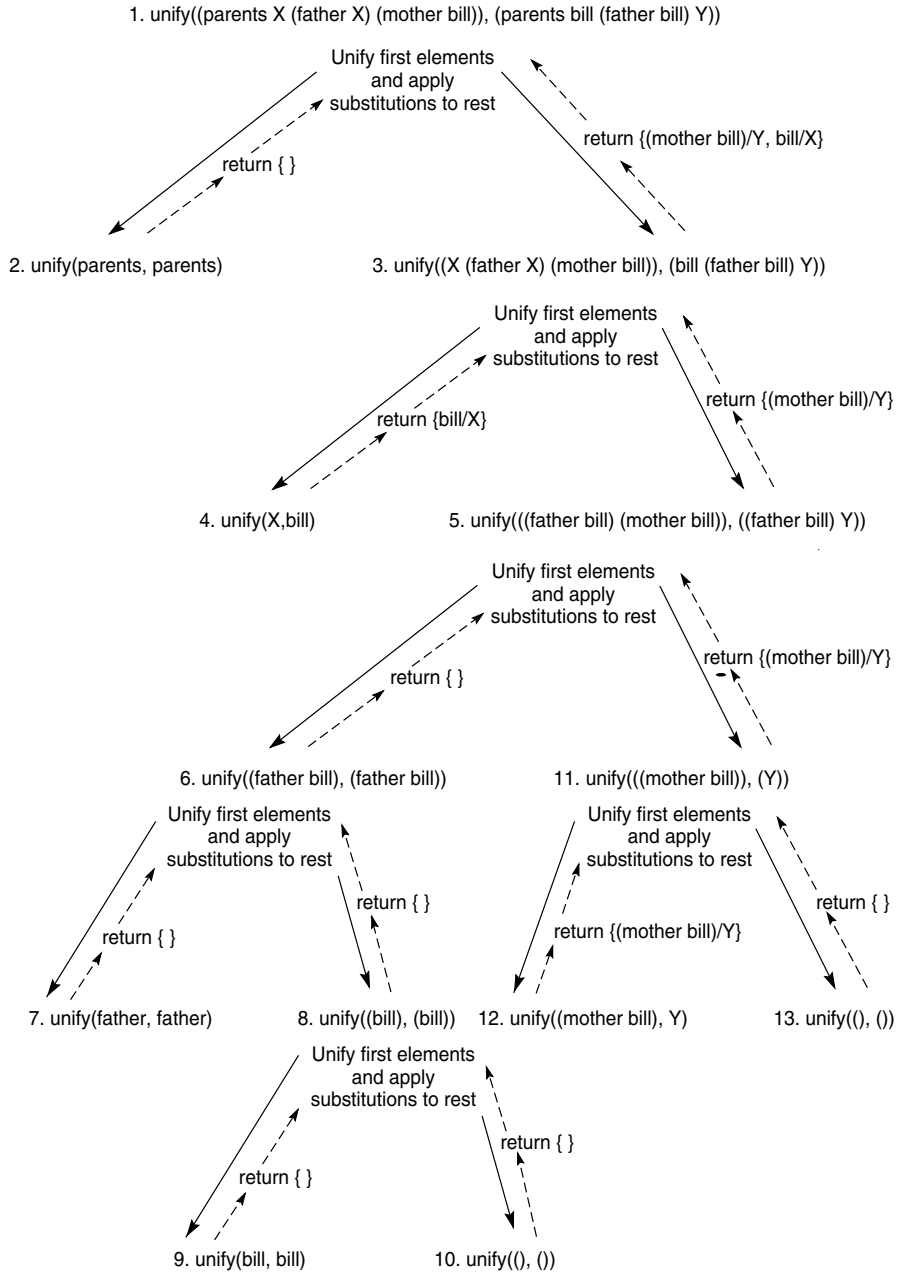


Figure 2.6 Final trace of the unification of (parents X (father X) (mother bill)) and (parents bill (father bill) Y).

2.4 Application: A Logic-Based Financial Advisor

As a final example of the use of predicate calculus to represent and reason about problem domains, we design a financial advisor using predicate calculus. Although this is a simple example, it illustrates many of the issues involved in realistic applications.

The function of the advisor is to help a user decide whether to invest in a savings account or the stock market. Some investors may want to split their money between the two. The investment that will be recommended for individual investors depends on their income and the current amount they have saved according to the following criteria:

1. Individuals with an inadequate savings account should always make increasing the amount saved their first priority, regardless of their income.
2. Individuals with an adequate savings account and an adequate income should consider a riskier but potentially more profitable investment in the stock market.
3. Individuals with a lower income who already have an adequate savings account may want to consider splitting their surplus income between savings and stocks, to increase the cushion in savings while attempting to increase their income through stocks.

The adequacy of both savings and income is determined by the number of dependents an individual must support. Our rule is to have at least $5,000 in savings for each dependent. An adequate income must be a steady income and supply at least $15,000 per year plus an additional $4,000 for each dependent.

To automate this advice, we translate these guidelines into sentences in the predicate calculus. The first task is to determine the major features that must be considered. Here, they are the adequacy of the savings and the income. These are represented by the predicates savings_account and income, respectively. Both of these are unary predicates, and their argument could be either adequate or inadequate. Thus,

```
savings_account(adequate).
savings_account(inadequate).
income(adequate).
income(inadequate).
```

are their possible values.

Conclusions are represented by the unary predicate investment, with possible values of its argument being stocks, savings, or combination (implying that the investment should be split).

Using these predicates, the different investment strategies are represented by implications. The first rule, that individuals with inadequate savings should make increased savings their main priority, is represented by

```
savings_account(inadequate) → investment(savings).
```

Similarly, the remaining two possible investment alternatives are represented by

savings_account(adequate) ∧ income(adequate) → investment(stocks).
savings_account(adequate) ∧ income(inadequate)
→ investment(combination).

Next, the advisor must determine when savings and income are adequate or inadequate. This will also be done using an implication. The need to do arithmetic calculations requires the use of functions. To determine the minimum adequate savings, the function minsavings is defined. minsavings takes one argument, the number of dependents, and returns 5000 times that argument.

Using minsavings, the adequacy of savings is determined by the rules

∀ X amount_saved(X) ∧ ∃ Y (dependents(Y) ∧ greater(X, minsavings(Y)))
→ savings_account(adequate).
∀ X amount_saved(X) ∧ ∃ Y (dependents(Y) ∧ ¬ greater(X, minsavings(Y)))
→ savings_account(inadequate).

where minsavings(X) ≡ 5000 * X.

In these definitions, amount_saved(X) and dependents(Y) assert the current amount in savings and the number of dependents of an investor; greater(X,Y) is the standard arithmetic test for one number being greater than another and is not formally defined in this example.

Similarly, a function minincome is defined as

minincome(X) ≡ 15000 + (4000 * X).

minincome is used to compute the minimum adequate income when given the number of dependents. The investor's current income is represented by a predicate, earnings. Because an adequate income must be both steady and above the minimum, earnings takes two arguments: the first is the amount earned, and the second must be equal to either steady or unsteady. The remaining rules needed for the advisor are

∀ X earnings(X, steady) ∧ ∃ Y (dependents(Y) ∧ greater(X, minincome(Y)))
→ income(adequate).

∀ X earnings(X, steady) ∧ ∃ Y (dependents(Y) ∧ ¬ greater(X, minincome(Y)))
→ income(inadequate).

∀ X earnings(X, unsteady) → income(inadequate).

In order to perform a consultation, a description of a particular investor is added to this set of predicate calculus sentences using the predicates amount_saved, earnings, and dependents. Thus, an individual with three dependents, $22,000 in savings, and a steady income of $25,000 would be described by

amount_saved(22000).

earnings(25000, steady).
dependents(3).

This yields a logical system consisting of the following sentences:

1. savings_account(inadequate) $\rightarrow$ investment(savings).
2. savings_account(adequate) $\wedge$ income(adequate) $\rightarrow$ investment(stocks).
3. savings_account(adequate) $\wedge$ income(inadequate) $\rightarrow$ investment(combination).
4. $\forall$ X amount_saved(X) $\wedge$ $\exists$ Y (dependents(Y) $\wedge$ greater(X, minsavings(Y))) $\rightarrow$ savings_account(adequate).
5. $\forall$ X amount_saved(X) $\wedge$ $\exists$ Y (dependents(Y) $\wedge$ $\neg$ greater(X, minsavings(Y))) $\rightarrow$ savings_account(inadequate).
6. $\forall$ X earnings(X, steady) $\wedge$ $\exists$ Y (dependents (Y) $\wedge$ greater(X, minincome(Y))) $\rightarrow$ income(adequate).
7. $\forall$ X earnings(X, steady) $\wedge$ $\exists$ Y (dependents(Y) $\wedge$ $\neg$ greater(X, minincome(Y))) $\rightarrow$ income(inadequate).
8. $\forall$ X earnings(X, unsteady) $\rightarrow$ income(inadequate).
9. amount_saved(22000).
10. earnings(25000, steady).
11. dependents(3).

where minsavings(X) $\equiv$ 5000 $*$ X and minincome(X) $\equiv$ 15000 + (4000 $*$ X).

This set of logical sentences describes the problem domain. The assertions are numbered so that they may be referenced in the following trace.

Using unification and modus ponens, a correct investment strategy for this individual may be inferred as a logical consequence of these descriptions. A first step would be to unify the conjunction of 10 and 11 with the first two components of the premise of 7; i.e.,

earnings(25000,steady) $\wedge$ dependents(3)

unifies with

earnings(X,steady) $\wedge$ dependents(Y)

under the substitution {25000/X, 3/Y}. This substitution yields the new implication:

earnings(25000, steady) $\wedge$ dependents(3) $\wedge$ $\neg$ greater(25000, minincome(3)) $\rightarrow$ income(inadequate).

Evaluating the function minincome yields the expression

earnings(25000, steady) ∧ dependents(3) ∧ ¬ greater(25000, 27000)
→ income(inadequate).

Because all three components of the premise are individually true, by 10, 3, and the mathematical definition of greater, their conjunction is true and the entire premise is true. Modus ponens may therefore be applied, yielding the conclusion income(inadequate). This is added as assertion 12.

12. income(inadequate).

Similarly,

amount_saved(22000) ∧ dependents(3)

unifies with the first two elements of the premise of assertion 4 under the substitution {22000/X, 3/Y}, yielding the implication

amount_saved(22000) ∧ dependents(3) ∧ greater(22000, minsavings(3))
→ savings_account(adequate).

Here, evaluating the function minsavings(3) yields the expression

amount_saved(22000) ∧ dependents(3) ∧ greater(22000, 15000)
→ savings_account(adequate).

Again, because all of the components of the premise of this implication are true, the entire premise evaluates to true and modus ponens may again be applied, yielding the conclusion savings_account(adequate), which is added as expression 13.

13. savings_account(adequate).

As an examination of expressions 3, 12, and 13 indicates, the premise of implication 3 is also true. When we apply modus ponens yet again, the conclusion is investment(combination), the suggested investment for our individual.

This example illustrates how predicate calculus may be used to reason about a realistic problem, drawing correct conclusions by applying inference rules to the initial problem description. We have not discussed exactly how an algorithm can determine the correct inferences to make to solve a given problem or the way in which this can be implemented on a computer. These topics are presented in Chapters 3, 4, and 6.

2.5 Epilogue and References

In this chapter we introduced predicate calculus as a representation language for AI problem solving. The symbols, terms, expressions, and semantics of the language were described and defined. Based on the semantics of predicate calculus, we defined inference rules that allow us to derive sentences that logically follow from a given set of expressions. We defined a unification algorithm that determines the variable substitutions that make two expressions match, which is essential for the application of inference rules. We concluded the chapter with the example of a financial advisor that represents financial knowledge with the predicate calculus and demonstrates logical inference as a problem-solving technique.

Predicate calculus is discussed in detail in a number of computer science books, including: *The Logical Basis for Computer Programming* by Zohar Manna and Richard Waldinger (1985), *Logic for Computer Science* by Jean H. Gallier (1986), *Symbolic Logic and Mechanical Theorem Proving* by Chin-liang Chang and Richard Char-tung Lee (1973), and *An Introduction to Mathematical Logic and Type Theory* by Peter B. Andrews (1986). We present more modern proof techniques in Chapter 14, Automated Reasoning.

Books that describe the use of predicate calculus as an artificial intelligence representation language include: *Logical Foundations of Artificial Intelligence* by Michael Genesereth and Nils Nilsson (1987), *Artificial Intelligence* by Nils Nilsson (1998), *The Field of Automated Reasoning* by Larry Wos (1995), *Computer Modelling of Mathematical Reasoning* by Alan Bundy (1983, 1988), and *Readings in Knowledge Representation* by Ronald Brachman and Hector Levesque (1985). See *Automated Reasoning* by Bob Veroff (1997) for interesting applications of automated inference. The *Journal for Automated Reasoning (JAR)* and *Conference on Automated Deduction (CADE)* cover current topics.

2.6 Exercises

1. Using truth tables, prove the identities of Section 2.1.2.
2. A new operator, $\oplus$, or *exclusive-or*, may be defined by the following truth table:

P	Q	P $\oplus$ Q
T	T	F
T	F	T
F	T	T
F	F	F

Create a propositional calculus expression using only $\wedge$, $\vee$, and $\neg$ that is equivalent to P $\oplus$ Q.
Prove their equivalence using truth tables.

3. The logical operator "↔" is read "if and only if." $P \leftrightarrow Q$ is defined as being equivalent to $(P \rightarrow Q) \wedge (Q \rightarrow P)$. Based on this definition, show that $P \leftrightarrow Q$ is logically equivalent to $(P \vee Q) \rightarrow (P \wedge Q)$:
   a. By using truth tables.
   b. By a series of substitutions using the identities on page 51.

4. Prove that implication is transitive in the propositional calculus, that is, that $((P \rightarrow Q) \wedge (Q \rightarrow R)) \rightarrow (P \rightarrow R)$.

5. a. Prove that modus ponens is sound for propositional calculus. Hint: use truth tables to enumerate all possible interpretations.
   b. *Abduction* is an inference rule that infers P from $P \rightarrow Q$ and Q. Show that abduction is not sound (see Chapter 7).
   c. Show modus tollens $((P \rightarrow Q) \wedge \neg Q) \rightarrow \neg P$ is sound.

6. Attempt to unify the following pairs of expressions. Either show their most general unifiers or explain why they will not unify.
   a. p(X,Y) and p(a,Z)
   b. p(X,X) and p(a,b)
   c. ancestor(X,Y) and ancestor(bill,father(bill))
   d. ancestor(X,father(X)) and ancestor(david,george)
   e. q(X) and $\neg$ q(a)

7. a. Compose the substitution sets {a/X, Y/Z} and {X/W, b/Y}.
   b. Prove that composition of substitution sets is associative.
   c. Construct an example to show that composition is not commutative.

8. Implement the unify algorithm of Section 2.3.2 in the computer language of your choice.

9. Give two alternative interpretations for the blocks world description of Figure 2.3.

10. Jane Doe has four dependents, a steady income of $30,000, and $15,000 in her savings account. Add the appropriate predicates describing her situation to the general investment advisor of the example in Section 2.4 and perform the unifications and inferences needed to determine her suggested investment.

11. Write a set of logical predicates that will perform simple automobile diagnostics (e.g., if the engine won't turn over and the lights won't come on, then the battery is bad). Don't try to be too elaborate, but cover the cases of bad battery, out of gas, bad spark plugs, and bad starter motor.

12. The following story is from N. Wirth's (1976) *Algorithms + data structures = programs*.

    I married a widow (let's call her W) who has a grown-up daughter (call her D). My father (F), who visited us quite often, fell in love with my step-daughter and married her. Hence my father became my son-in-law and my step-daughter became my mother. Some months later, my wife gave birth to a son (S_1), who became the brother-in-law of my father, as well as my uncle. The wife of my father, that is, my step-daughter, also had a son (S_2).

    Using predicate calculus, create a set of expressions that represent the situation in the above story. Add expressions defining basic family relationships such as the definition of father-in-law and use modus ponens on this system to prove the conclusion that "I am my own grandfather."