3.1 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.

3.1.2 Predicate calculus symbols

Predicate calculus symbol can be any derived from any of the following sets: **set of letter, set of digit** and the **underscore (_)**. Any predicate calculus should symbol start with a letter. It may represent a **variable**, **a constant**, a **function** or **predicate**.

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 **Ali**, **ISMAEL**, and **KAte** are legal variables, whereas **aLI** and **ismael** are not.

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.

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 domain into unique element in the range Elements of the domain and range are objects in the world of discourse for instance father (ibrahim) → abdul for the father of ibrahim yielding the result abdul.

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.

D E F I N I T I O N

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,t1, t2,…,tn enclosed in parentheses and separated by commas.

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

3.1.3 Predicate logic and quantifiers

Predicate logic (also called first order predicate logic) has a similar formalism like propositional logic. However, the capability of reasoning and knowledge representation using predicate logic is higher than propositional logic. For instance, it includes two more quantifiers, namely, the essential quantifier (∀) and the existential quantifier (). To illustrate the use of the quantifiers, let us consider the following pieces of knowledge.

Knowledge 1 : ***All boys like sweets.***

Using predicate logic, we can write the above statement as

$$\forall X (Boy (X) \rightarrow Likes (X , sweets))$$

Knowledge 2 : ***Some boys like flying kites.***

Using predicate logic, the above statement can be represented as

$$\exists X (Boy (X) \rightarrow Likes (X, Flying\text{-}kites))$$

You must know the following things

For predicates p and q and variables X and Y:

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

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

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

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

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

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

3.1.4 Semantics for 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.

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, ∀ :

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

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 ∀ 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.

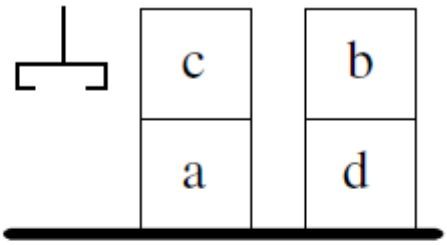
"Blocks World" Example of Semantic Meaning

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 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. 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.



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

Figure: block world with its predicate calculus description

To pick up **a** block and stack it on another block, both blocks must be clear. In Figure, 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:

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

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.

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

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.

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.

3.2 Examples of English sentences represented in predicate calculus

1. If it doesn't rain on Monday, Tom will go to the mountains.
   $\neg weather(rain, monday) \rightarrow go(tom, mountains)$
2. Emma is a Doberman pinscher and a good dog.
   $gooddog(emma) \wedge isa(emma, doberman)$
3. All basketball players are tall.
   $\forall X (basketball\_player(X) \rightarrow tall(X))$
4. Some people like anchovies.
   $\exists X (person(X) \wedge likes(X, anchovies))$

5. If wishes were horses, beggars would ride.
   equal(wishes, horses)→ride(beggars)
6. Nobody likes taxes.
   ¬ ∃ X likes(X, taxes)
7. Everybody loves somebody
   $\forall x \ \exists y \ \ Loves(x, y)$.
8. There is some one who is loved by everyone
   $\exists y \ \forall x \ \ Loves(x, y)$.

The order of the quantification is very important

9. Everyone dislikes parsnips

The negation effect over quantification is very important

$\forall x \ \neg Likes(x, Parsnips)$ is equivalent to $\neg \exists x \ \ Likes(x, Parsnips)$.

10. Everyone like ice cream

$\forall x \ \ Likes(x, IceCream)$ is equivalent to $\neg \exists x \ \neg Likes(x, IceCream)$.

Example : Rewrite the following sentences in FOL.

1. Coconut-crunchy is a biscuit.
2. Mary is a child who takes coconut-crunchy.
3. John loves children who take biscuits.
4. John loves Mary.

The above statements can be represented in FOL using two quantifiers X & Y.

1. Biscuit (coconut-crunchy)

2. Child (mary) ∧ Takes (mary, coconut-crunchy)

3. ∀X ((Child (X) ∧ ∃Y (Takes (X, Y) ∧ Biscuit (Y))) →Loves (john, X)

---

4. Loves (john, mary)

Example: represent the following sentence in first order logic using consistent vocabulary (which must be define)

a) Some student took French in spring 2001.

b. Every student who takes French passes it.

c. Only one student took Greek in spring 2001.

d. The best score in Greek is always higher than the best score in French.

e. Every person who buys a policy is smart.

f. No person buys an expensive policy.

g. There is an agent who sells policies only to people who are not insured.

h. There is a barber who shaves all men in town who do not shave themselves.

i. A person born in the UK, each of whose parents is a UK citizen or a UK resident, is a UK citizen by birth.

j. A person born outside the UK, one of whose parents is a UK citizen by birth, is a UK citizen by descent.

k. Politicians can fool some of the people all of the time, and they can fool all of the people some of the time, but they can't fool all of the people all of the time.

nectives and quantifiers and the use of predicates, functions, constants, and equality. Let the basic vocabulary be as follows:
$Takes(x, c, s)$: student x takes course c in semester s;
$Passes(x, c, s)$: student x passes course c in semester s;
$Score(x, c, s)$: the score obtained by student x in course c in semester s;
$x > y$: x is greater than y;
F and G: specific French and Greek courses (one could also interpret these sentences as referring to *any* such course, in which case one could use a predicate $Subject(c, f)$ meaning that the subject of course c is field f;
$Buys(x, y, z)$: x buys y from z (using a binary predicate with unspecified seller is OK but less felicitous);
$Sells(x, y, z)$: x sells y to z;
$Shaves(x, y)$: person x shaves person y
$Born(x, c)$: person x is born in country c;
$Parent(x, y)$: x is a parent of y;
$Citizen(x, c, r)$: x is a citizen of country c for reason r;
$Resident(x, c)$: x is a resident of country c;
$Fools(x, y, t)$: person x fools person y at time t;
$Student(x)$, $Person(x)$, $Man(x)$, $Barber(x)$, $Expensive(x)$, $Agent(x)$, $Insured(x)$, $Smart(x)$, $Politician(x)$: predicates satisfied by members of the corresponding categories.

a. Some students took French in spring 2001.
$\exists x \;\; Student(x) \wedge Takes(x, F, Spring2001).$

b. Every student who takes French passes it.
$\forall x, s \;\; Student(x) \wedge Takes(x, F, s) \;\Rightarrow\; Passes(x, F, s).$

c. Only one student took Greek in spring 2001.
$\exists x \;\; Student(x) \wedge Takes(x, G, Spring2001) \wedge \forall y \;\; y \neq x \Rightarrow \neg Takes(y, G, Spring2001).$

d. The best score in Greek is always higher than the best score in French.
$\forall s \;\; \exists x \;\; \forall y \;\; Score(x, G, s) > Score(y, F, s).$

e. Every person who buys a policy is smart.
$\forall x \;\; Person(x) \wedge (\exists y, z \;\; Policy(y) \wedge Buys(x, y, z)) \;\Rightarrow\; Smart(x).$

f. No person buys an expensive policy.
$\forall x, y, z \;\; Person(x) \wedge Policy(y) \wedge Expensive(y) \;\Rightarrow\; \neg Buys(x, y, z).$

g. There is an agent who sells policies only to people who are not insured.
$\exists x \;\; Agent(x) \wedge \forall y, z \;\; Policy(y) \wedge Sells(x, y, z) \;\Rightarrow\; (Person(z) \wedge \neg Insured(z)).$

h. There is a barber who shaves all men in town who do not shave themselves.
$\exists x \;\; Barber(x) \wedge \forall y \;\; Man(y) \wedge \neg Shaves(y, y) \;\Rightarrow\; Shaves(x, y).$

i. A person born in the UK, each of whose parents is a UK citizen or a UK resident, is a UK citizen by birth.
$\forall x \;\; Person(x) \wedge Born(x, UK) \wedge (\forall y \;\; Parent(y, x) \Rightarrow ((\exists r \;\; Citizen(y, UK, r)) \vee Resident(y, UK))) \;\Rightarrow\; Citizen(x, UK, Birth).$

j. A person born outside the UK, one of whose parents is a UK citizen by birth, is a UK

$\forall x \;\; Person(x) \wedge \neg Born(x, UK) \wedge (\exists y \;\; Parent(y, x) \wedge Citizen(y, UK, Birth)) \;\Rightarrow\; Citizen(x, UK, Descent).$

k. Politicians can fool some of the people all of the time, and they can fool all of the people some of the time, but they can't fool all of the people all of the time.

$$\begin{aligned}&\forall x \;\; Politician(x) \;\Rightarrow\\ &\quad (\exists y \;\; \forall t \;\; Person(y) \wedge Fools(x, y, t)) \wedge\\ &\quad (\exists t \;\; \forall y \;\; Person(y) \;\Rightarrow\; Fools(x, y, t)) \wedge\\ &\quad \neg(\forall t \;\; \forall y \;\; Person(y) \;\Rightarrow\; Fools(x, y, t))\end{aligned}$$

3.3 Using inference rule to produce predicate calculus expressions

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.
Modus ponens and a number of other useful inference rules are defined below.
D E F I N I T I O N
MODUS PONENS, MODUS TOLLENS, AND ELIMINATION, AND

INTRODUCTION and UNIVERSAL INSTANTIATION

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

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

And elimination allows us to infer the truth of either of the conjuncts from the truth of a conjunctive sentence. For instance, P ∧ 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 ∧ 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, ∀ 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 → Q. Because it is indeed now raining (P is true), our set of axioms becomes

P→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:

∀ X (man(X)→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)→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*. More powerful rule of inference called *resolution*, which is the basis of many automated reasoning systems.

3.4 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)).

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 X1 is substituted for another variable X2 and at a later time X1 is replaced by a constant, then X2 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.

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.

D E F I N I T I O N

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))

```
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
```

3.4.1 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.

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

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(( ), ( ))

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 thesubstitution {(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 . 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.

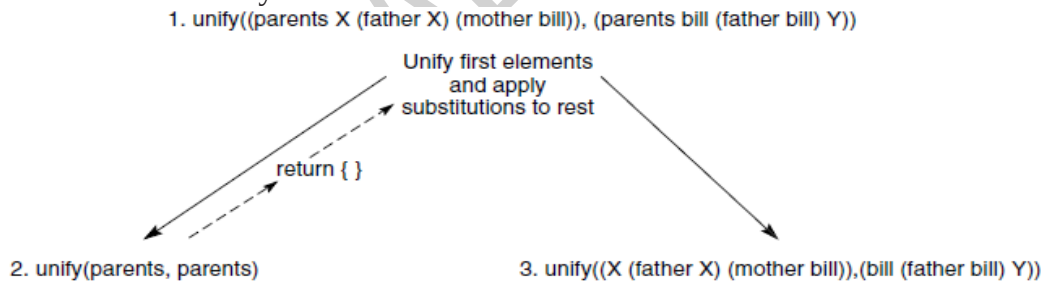


Figure Initial steps in the unification of (parents X (father X) (mother bill)) and (parents bill (father bill) Y).

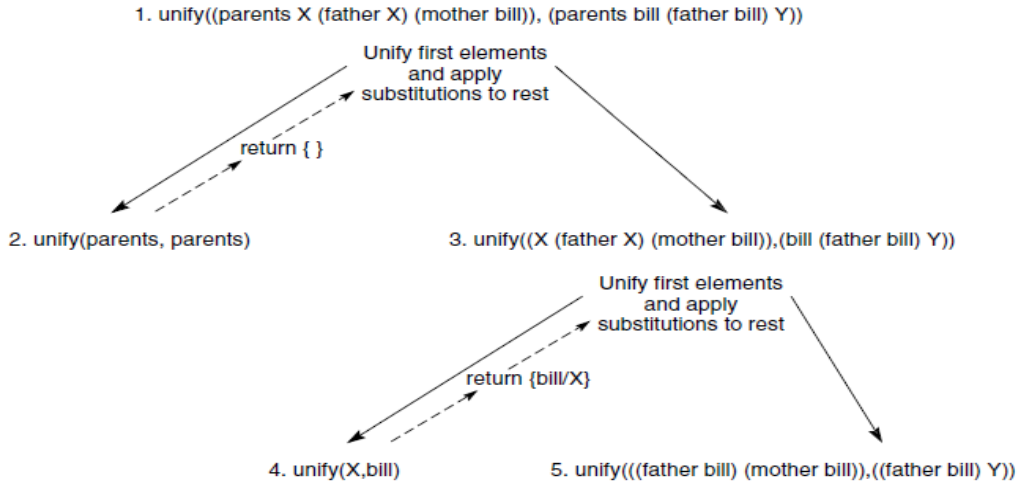
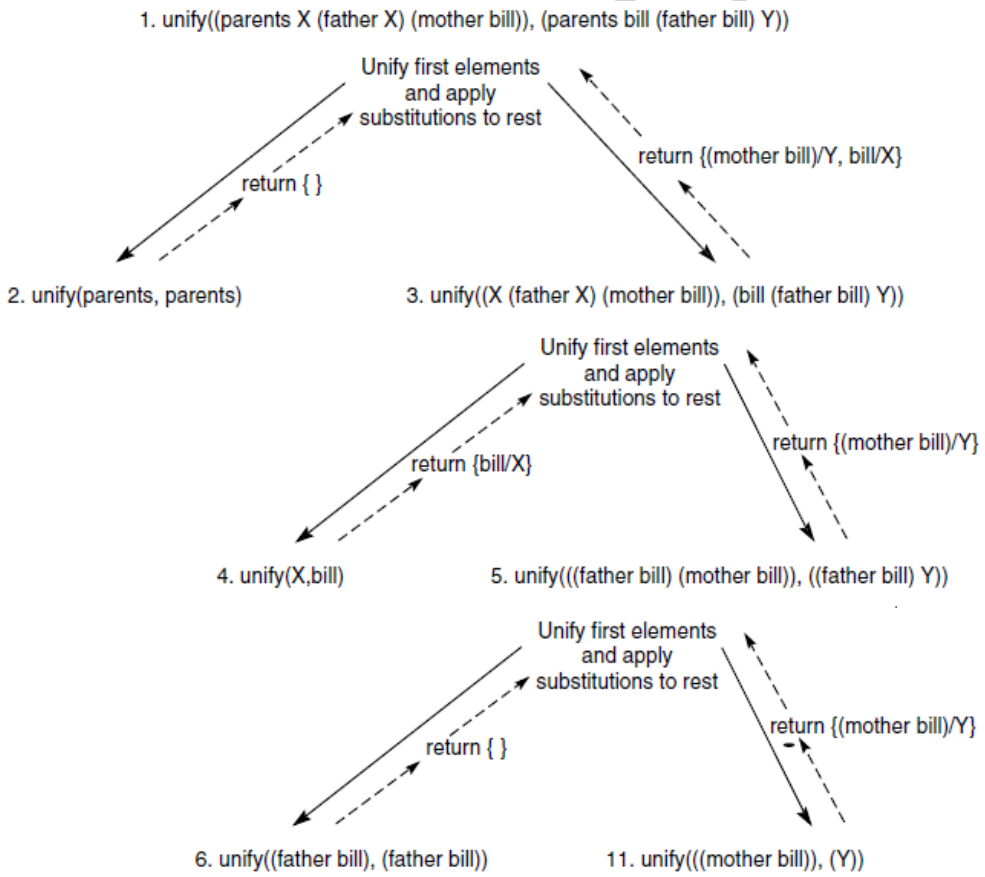


Figure Further steps in the unification of (parents X (father X) (mother bill)) and (parents bill (father bill) Y).



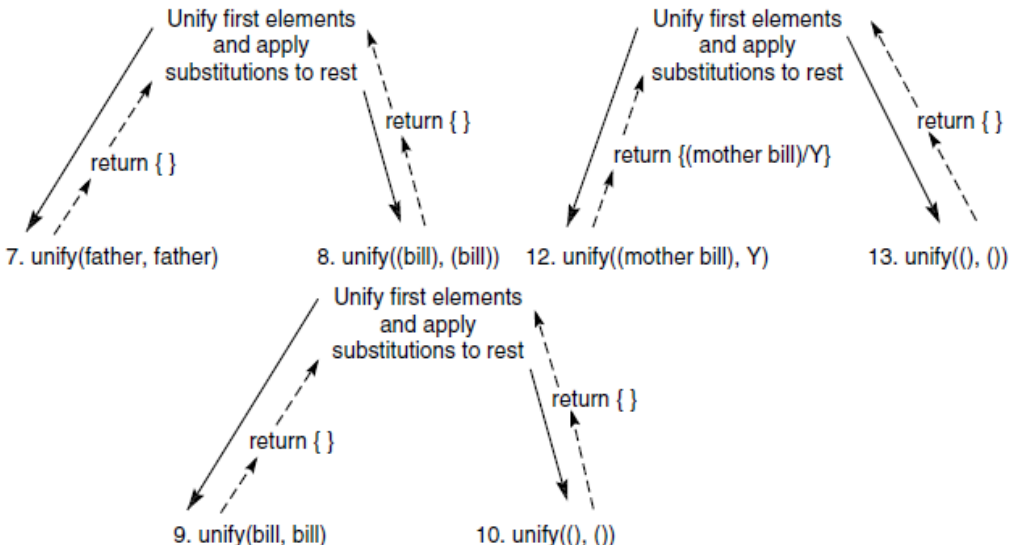


Figure Final trace of the unification of (parents X (father X) (mother bill)) and (parents bill (father bill) Y).