## UNIT III KNOWLEDGE REPRESENTATION

First Order Predicate Logic – Prolog Programming – Unification – Forward Chaining- Backward Chaining – Resolution – Knowledge Representation - Ontological Engineering-Categories and Objects – Events - Mental Events and Mental Objects - Reasoning Systems for Categories - Reasoning with Default Information

### FIRST ORDER PREDICATE LOGIC

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation.

**Syntax and Semantics of First Order Predicate Logic**

The **domain** of a model is DOMAIN the set of objects or **domain elements** it contains. The domain is required to be *nonempty*—every possible world must contain at least one object. Figure 8.2 shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.The objects in the model may be *related* in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation TUPLE is just the set of **tuples** of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set

*{ ⟨Richard the Lionheart, King John⟩, ⟨ King John, Richard the Lionheart⟩ }*

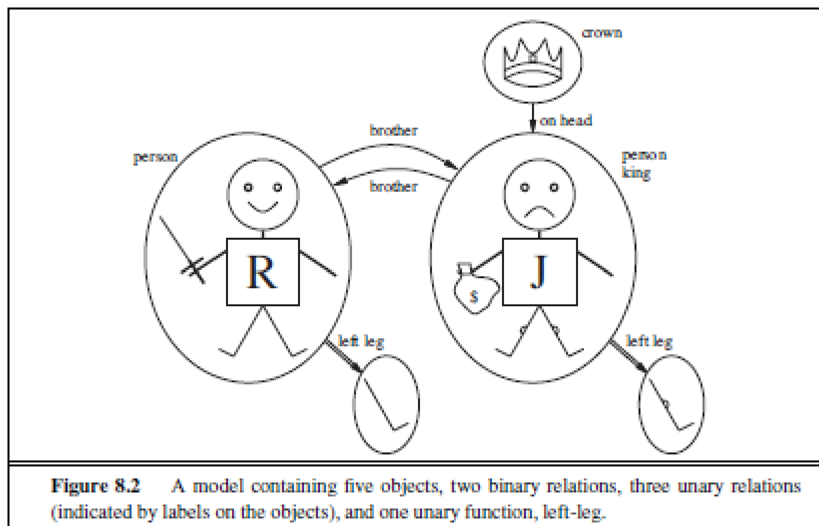



**Figure 8.2** A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

The crown is on King John's head, so the "on head" relation contains just one tuple, _the crown, King John_. The "brother" and "on head" relations are binary relations—that is, they relate pairs of objects. Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:

*⟨Richard the Lionheart⟩ → Richard's left leg*
*⟨King John⟩ → John's left leg*

**Symbols and interpretations**

We turn now to the syntax of first-order logic . The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: **constant symbols**,which stand for objects; **predicate symbols**, which

stand for relations; and **function symbols**, which stand for functions. We adopt the convention that these symbols will begin with uppercase letters. For example, we might use the constant symbols Richard and John; the predicate symbols Brother , OnHead, Person, King, and Crown; and the function symbol LeftLeg.

Each model includes an **interpretation** that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols. One possible interpretation for our example is as follows

- Richard refers to Richard the Lionheart and John refers to the evil King John.
- Brother refers to the brotherhood relation, that is, the set of tuples of objects; OnHead refers to the "on head" relation that holds between the crown and King John; Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns.
- LeftLeg refers to the "left leg" function

A complete description from the formal grammar is as follows

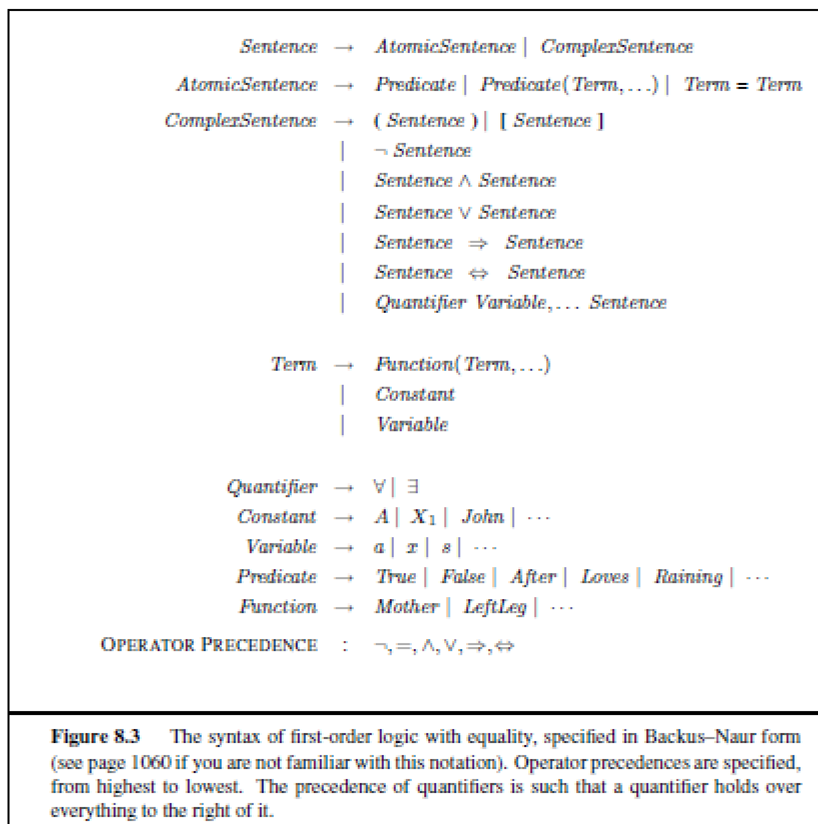



**Figure 8.3** The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

**Term**

A term is a logical expression that refers TERM to an object. Constant symbols are therefore terms,
but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use *LeftLeg(John)*. The formal semantics of terms is straightforward. Consider a term f(t1, . . . , tn). The function symbol f refers to some function in the model.

**Atomic sentences**

Atomic sentence (or atom for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as *Brother (Richard , John).*

Atomic sentences can have complex terms as arguments. Thus,

*Married(Father (Richard), Mother (John))* states that Richard the Lionheart's father is married to King John's mother.

**Complex Sentences**

We can use **logical connectives** to construct more complex sentences, with the same syntax and semantics as in propositional calculus

$\neg Brother(LeftLeg(Richard), John)$

$Brother(Richard, John) \land Brother(John, Richard)$

$King(Richard) \lor King(John)$

$\neg King(Richard) \Rightarrow King(John)$.

**Quantifiers**

Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name. First-order logic contains two standard quantifiers, called *universal* and *existential*

**Universal quantification ( ∀ )**

"All kings are persons," is written in first-order logic as

$\forall x\ King(x) \Rightarrow Person(x)$

∀ is usually pronounced "For all . . . " Thus, the sentence says, "For all $x$, if $x$ is a king, then $x$ is a person." The symbol $x$ is called a **variable**. A term with no variables is called a **ground term**.

Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

$x \rightarrow$ *Richard the Lionheart,*

$x \rightarrow$ *King John,*

$x \rightarrow$ *Richard's left leg,*

$x \rightarrow$ *John's left leg,*

$x \rightarrow$ *the crown.*

The universally quantified sentence $\forall x\ King(x) \Rightarrow Person(x)$ is true in the original model if the sentence $King(x) \Rightarrow Person(x)$ is true under each of the five extended interpretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

*Richard the Lionheart is a king* ⇒ *Richard the Lionheart is a person.*

*King John is a king* ⇒ *King John is a person.*

*Richard's left leg is a king* ⇒ *Richard's left leg is a person.*

*John's left leg is a king* ⇒ *John's left leg is a person.*

*The crown is a king* ⇒ *the crown is a person.*

**Existential quantification ( ∃ )**

Universal quantification makes statements about every object. Similarly, we can make a statement
about *some* object in the universe without naming it, by using an existential quantifier.
To say, for example, that King John has a crown on his head, we write

$\exists x\ Crown(x) \land OnHead(x, John)$.

$\exists x$ is pronounced "There exists an $x$ such that . . . " or "For some $x$ . . . "

More precisely, $\exists x\ P$ is true in a given model if $P$ is true in *at least one* extended interpretation

that assigns $x$ to a domain element. That is, at least one of the following is true:
*Richard the Lionheart is a crown* $\wedge$ *Richard the Lionheart is on John's head;*
*King John is a crown* $\wedge$ *King John is on John's head;*
*Richard's left leg is a crown* $\wedge$ *Richard's left leg is on John's head;*
*John's left leg is a crown* $\wedge$ *John's left leg is on John's head;*
*The crown is a crown* $\wedge$ *the crown is on John's head.*
The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model.
Just as $\Rightarrow$ appears to be the natural connective to use with $\forall$, $\wedge$ is the natural connective to use with $\exists$.
Using $\wedge$ as the main connective with $\forall$ led to an overly strong statement in the example in the previous section; using $\Rightarrow$ with $\exists$ usually leads to a very weak statement, indeed. Consider the following sentence:

$$\exists\ x\ Crown(x) \Rightarrow OnHead(x,\ John)\ .$$

Applying the semantics, we see that the sentence says that at least one of the following assertions is true:
*Richard the Lionheart is a crown* $\Rightarrow$ *Richard the Lionheart is on John's head;*
*King John is a crown* $\Rightarrow$ *King John is on John's head;*
*Richard's left leg is a crown* $\Rightarrow$ *Richard's left leg is on John's head;*
and so on. Now an implication is true if both premise and conclusion are true, *or if its premise is false*. So if Richard the Lionheart is not a crown, then the first assertion is true and the existential is satisfied. So, an existentially quantified implication sentence is true whenever *any* object fails to satisfy the premise

**Nested quantifiers**
We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "*Brothers are siblings*" can be written as

$$\forall\ x\ \forall\ y\ Brother\ (x,\ y) \Rightarrow Sibling(x,\ y)$$

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

$$\forall\ x,\ y\ Sibling(x,\ y) \Leftrightarrow Sibling(y,\ x)\ .$$

In other cases we will have mixtures. "*Everybody loves somebody*" means that for every person, there is someone that person loves:

$$\forall\ x\ \exists\ y\ Loves(x,\ y)\ .$$

On the other hand, to say "*There is someone who is loved by everyone*," we write

$$\exists\ y\ \forall\ x\ Loves(x,\ y)\ .$$

The order of quantification is therefore very important. It becomes clearer if we insert parentheses.
$\forall\ x\ (\exists\ y\ Loves(x,\ y))$ says that *everyone* has a particular property, namely, the property that they love someone. On the other hand, $\exists\ y\ (\forall\ x\ Loves(x,\ y))$ says that *someone* in the world has a particular property, namely the property of being loved by everybody.

**Connections between $\forall$ and $\exists$**
The two quantifiers are actually intimately connected with each other, through negation. Asserting
that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$\forall x \neg Likes(x, Parsnips)$ is equivalent to $\neg\exists x\ Likes(x, Parsnips)$ .
We can go one step further: "*Everyone likes ice cream*" means that there is no one who does not like ice cream:

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

**Equality**
We can use the **equality symbol** to signify that two terms refer to the same object. For example,

$Father(John)=Henry$

says that the object referred to by $Father(John)$ and the object referred to by $Henry$ are the same.
The equality symbol can be used to state facts about a given function, as we just did for the $Father$ symbol. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, we would write

$\exists x, y\ Brother(x, Richard) \wedge Brother(y, Richard) \wedge \neg(x=y)$ .

**<u>USING FIRST ORDER LOGIC</u>**
Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions**. For example, we can assert that John is a king, Richard is a person, and all kings are persons:

*TELL*$(KB, King(John))$

*TELL*$(KB, Person(Richard))$

*TELL*$(KB, \forall x\ King(x) \Rightarrow Person(x))$

We can ask questions of the knowledge base using ASK. For example,

*ASK*$(KB, King(John))$

returns $true$. Questions asked with ASK are called **queries** or **goals**
If we want to know what value of $x$ makes the sentence true, we will need a different function, ASKVARS, which we call with

*ASKVARS*$(KB, Person(x))$

and which yields a stream of answers. In this case there will be two answers: $\{x/John\}$ and $\{x/Richard\}$. Such an answer is called a **substitution** or **binding list**.

**The kinship domain**
The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William" and rules such as "One's grandmother is the mother of one's parent."
Clearly, the objects in our domain are people. We have two unary predicates, Male and Female. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother , Sister , Child , Daughter, Son, Spouse, Wife, Husband, Grandparent , Grandchild , Cousin, Aunt, and Uncle. We use functions for Mother and Father , because every person has exactly one of each of these.
For example, one's mother is one's female parent:

$\forall m, c\ Mother(c)=m \Leftrightarrow Female(m) \wedge Parent(m, c)$ .

One's husband is one's male spouse:

$\forall w, h\ Husband(h, w) \Leftrightarrow Male(h) \wedge Spouse(h, w)$ .

Male and female are disjoint categories:

$\forall\ x\ Male(x) \Leftrightarrow \neg Female(x)$ .

Parent and child are inverse relations:

$\forall\ p,\ c\ Parent(p,\ c) \Leftrightarrow Child\ (c,\ p)$ .

A grandparent is a parent of one's parent:

$\forall\ g,\ c\ Grandparent\ (g,\ c) \Leftrightarrow \exists p\ Parent(g,\ p) \land Parent(p,\ c)$ .

A sibling is another child of one's parents:

$\forall\ x,\ y\ Sibling(x,\ y) \Leftrightarrow x _= y \land \exists p\ Parent(p,\ x) \land Parent(p,\ y)$ .

Each of these sentences can be viewed as an **axiom** of the kinship domain. Axioms are commonly associated with purely mathematical domains. Our kinship axioms are also **definitions**; they have the form $\forall\ x,\ y\ P(x,\ y) \Leftrightarrow \ldots\ldots$ The axioms define the Mother function and the Husband, Male, Parent, Grandparent, and Sibling predicates in terms of other predicates.

For example, consider the assertion that siblinghood is symmetric:

$\forall\ x,\ y\ Sibling(x,\ y) \Leftrightarrow Sibling(y,\ x)$ .

**The WUMPUS world**

The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence. A sample wumpus world is shown in Figure 7.2.

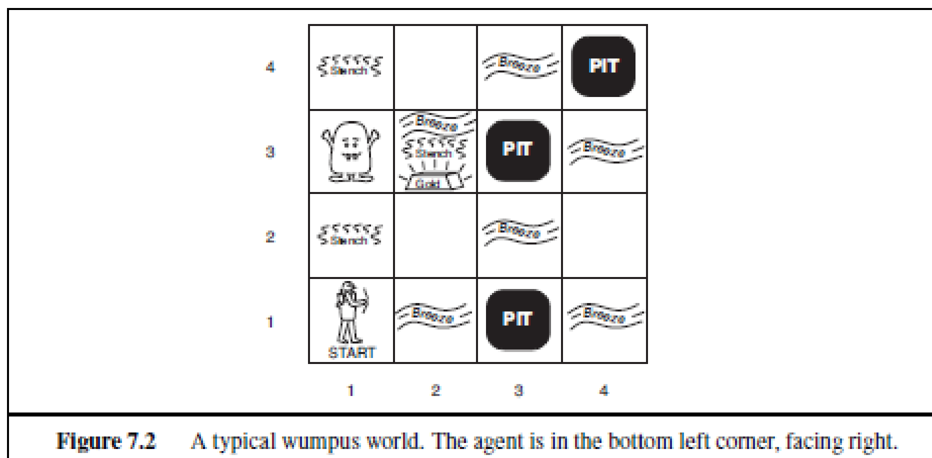



**Figure 7.2** A typical wumpus world. The agent is in the bottom left corner, facing right.

The wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what. We use integers for time steps. A typical percept sentence would be

$Percept\ ([Stench,\ Breeze,\ Glitter,\ None,\ None],\ 5)$ .

Here, Percept is a binary predicate, and Stench and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

$Turn(Right),\ Turn(Left),\ Forward,\ Shoot,\ Grab,\ Climb$ .

To determine which is best, the agent program executes the query

$ASKVARS(\exists\ a\ BestAction(a,\ 5))$ ,

which returns a binding list such as {a/Grab}. The agent program can then return Grab as the action to take. The raw percept data implies certain facts about the current state. For example:

$\forall t, s, g, m, c\ Percept([s, Breeze, g, m, c], t) \Rightarrow Breeze(t)$,

$\forall t, s, b, m, c\ Percept([s, b, Glitter, m, c], t) \Rightarrow Glitter(t)$

These rules exhibit a trivial form of the reasoning process called **perception.**
Simple "reflex" behavior can also be implemented by quantified implication sentences. For example, we have

$\forall\ t\ Glitter(t) \Rightarrow BestAction(Grab, t)$.

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion

BestAction(Grab, 5)—that is, Grab is the right thing to do.

For example, if the agent is at a square and perceives a breeze, then that square is breezy:

$\forall\ s, t\ At(Agent, s, t) \land Breeze(t) \Rightarrow Breezy(s)$.

It is useful to know that a *square* is breezy because we know that the pits cannot move about. Notice that Breezy has no time argument.
Having discovered which places are breezy (or smelly) and, very important, *not* breezy (or *not* smelly), the agent can deduce where the pits are (and where the wumpus is). first-order logic just needs one axiom:

$\forall\ s\ Breezy(s) \Leftrightarrow \exists r\ Adjacent(r, s) \land Pit(r)$.

**SUBSTITUTION**
Let us begin with universal quantifiers

$\forall\ x\ King(x) \land Greedy(x) \Rightarrow Evil(x)$.

Then it seems quite permissible to infer any of the following sentences:

$King(John) \land Greedy(John) \Rightarrow Evil(John)$

$King(Richard) \land Greedy(Richard) \Rightarrow Evil(Richard)$

$King(Father(John)) \land Greedy(Father(John)) \Rightarrow Evil(Father(John))$.

The rule of **Universal Instantiation** (**UI** for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable. Let SUBST($\theta$, $\alpha$) denote the result of applying the substitution $\theta$ to the sentence $\alpha$. Then the rule is written

$\forall v\ \alpha\ SUBST(\{v/g\}, \alpha)$

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

In the rule for **Existential Instantiation**, the variable is replaced by a single *new constant symbol*. The formal statement is as follows: for any sentence $\alpha$, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$\exists v\ \alpha\ SUBST(\{v/k\}, \alpha)$

For example, from the sentence

$\exists\ x\ Crown(x) \land OnHead(x, John)$

we can infer the sentence

$Crown(C1) \land OnHead(C1, John)$

**EXAMPLE**
Suppose our knowledge base contains just the sentences

$\forall x \; King(x) \wedge Greedy(x) \Rightarrow Evil(x)$

$King(John)$

$Greedy(John)$

$Brother(Richard, John)$

Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case, $\{x/John\}$ and $\{x/Richard\}$. We obtain

$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$

$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$,

and we discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences

$King(John)$,

$Greedy(John)$, and so on—as proposition symbols.

**UNIFICATION**

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them if one exists:

$UNIFY(p, q) = \theta$ where $SUBST(\theta, p) = SUBST(\theta, q)$.

Suppose we have a query

$AskVars(Knows(John, x))$: whom does John know? Answers can be found by finding all sentences in the knowledge base that unify with $Knows(John, x)$. Here are the results of unification with four different sentences that might be in the knowledge base:

$UNIFY(Knows(John, x), Knows(John, Jane)) = \{x/Jane\}$

$UNIFY(Knows(John, x), Knows(y, Bill)) = \{x/Bill, y/John\}$

$UNIFY(Knows(John, x), Knows(y, Mother(y))) = \{y/John, x/Mother(John)\}$

$UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail$.

The last unification fails because $x$ cannot take on the values John and Elizabeth at the same time. Now, remember that $Knows(x, Elizabeth)$ means "Everyone knows Elizabeth," so we *should* be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, $x$. The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes.

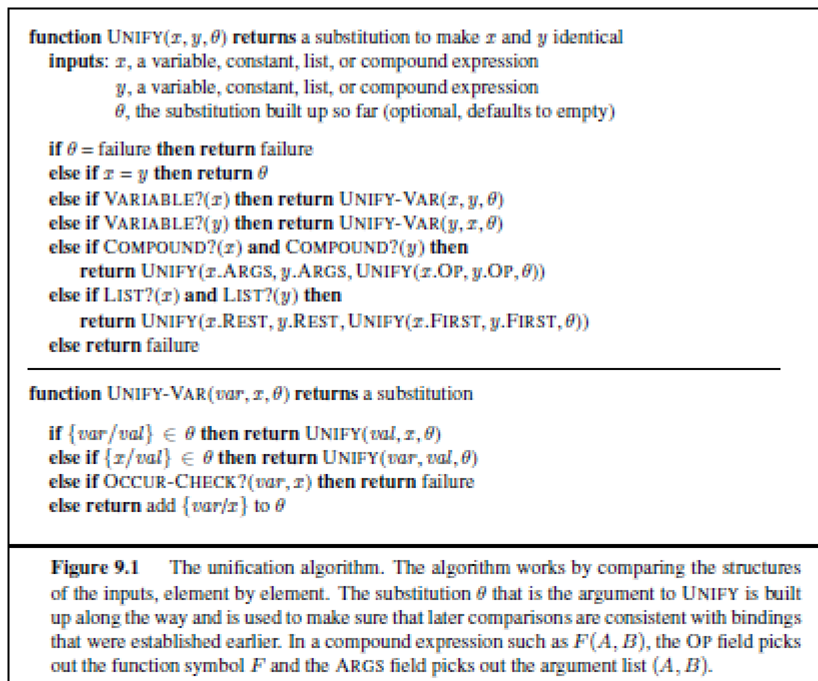
For example, we can rename $x$ in $Knows(x, Elizabeth)$ to $x17$ (a new variable name) without changing its meaning. Now the unification will work:

$UNIFY(Knows(John, x), Knows(x17, Elizabeth)) = \{x/Elizabeth, x17/John\}$

UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier. For example, $UNIFY(Knows(John, x), Knows(y, z))$ could return $\{y/John, x/z\}$ or $\{y/John, x/John, z/John\}$. The first unifier gives $Knows(John, z)$ as the result of unification, whereas the second gives

*Knows(John, John)*. The second result could be obtained from the first by an additional substitution {z/John} ; we say that the first unifier is *more general* than the second, because it places fewer restrictions on the values of the variables.

An algorithm for computing most general unifiers is shown in Figure 9.1. The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed



```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure

function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

**Figure 9.1** The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution $\theta$ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol $F$ and the ARGS field picks out the argument list $(A, B)$.

## FORWARD CHAINING

The following are first-order definite clauses:

*King(x) ∧ Greedy(x) ⇒ Evil(x)*

*King(John)*

*Greedy(y)*

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. Consider the following problem:

***The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by ColonelWest, who is American.***

We will prove thatWest is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

". . . it is a crime for an American to sell weapons to hostile nations":

*American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal (x)* ---**(9.3)**

"Nono . . . has some missiles." The sentence ∃ x Owns(Nono, x) ∧ Missile(x) is transformed

into two definite clauses by Existential Instantiation, introducing a new constant M1:

*Owns(Nono, M1)*-------------------------------------------------------------------------------**(9.4)**

$Missile(M1)$ ------ *(9.5)*

"All of its missiles were sold to it by Colonel West":

$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$ ------ *(9.6)*

We will also need to know that missiles are weapons:

$Missile(x) \Rightarrow Weapon(x)$ ------ *(9.7)*

and we must know that an enemy of America counts as "hostile":

$Enemy(x, America) \Rightarrow Hostile(x)$ ------------------------------------------------------ *(9.8)*

"West, who is American ..........":

$American(West)$ ------------------------------------------------------------------------ *(9.9)*

The country Nono, an enemy of America ......... ":

$Enemy(Nono, America)$ ------------------------------------------------------------------- *(9.10)*

This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases. Datalog DATALOG is a language that is restricted to first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases.

**<u>A simple Forward-Chaining Algorithm</u>**

The first forward-chaining algorithm we consider is a simple one, shown in Figure 9.3. Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered (assuming that just one answer is required) or no new facts are added. We use our crime problem to illustrate how FOL-FC-ASK works. The implication
sentences are (9.3), (9.6), (9.7), and (9.8). Two iterations are required:
On the first iteration, rule (9.3) has unsatisfied premises.

Rule (9.6) is satisfied with $\{x/M1\}$, *and* $Sells(West, M1, Nono)$ is added.

Rule (9.7) is satisfied with $\{x/M1\}$, *and* $Weapon(M1)$ is added.

Rule (9.8) is satisfied with $\{x/Nono\}$, *and* $Hostile(Nono)$ is added.

On the second iteration, rule (9.3) is satisfied with $\{x/West, y/M1, z/Nono\}$, and $Criminal(West)$ is added.

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
      new ← { }
      for each rule in KB do
          (p1 ∧ . . . ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
          for each θ such that SUBST(θ, p1 ∧ . . . ∧ pn) = SUBST(θ, p'1 ∧ . . . ∧ p'n)
                      for some p'1, . . . , p'n in KB
              q' ← SUBST(θ, q)
              if q' does not unify with some sentence already in KB or new then
                  add q' to new
                  φ ← UNIFY(q', α)
                  if φ is not fail then return φ
      add new to KB
  return false
```

**Figure 9.3** A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to $KB$ all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in $KB$. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

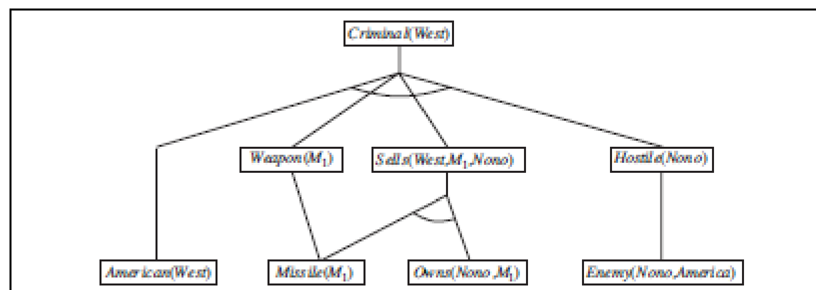



**Figure 9.4** The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Figure 9.4 shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a **fixed point** of the inference process. Fixed points reached by forward chaining with first-order definite clauses are similar to those for propositional forward chaining.

FOL-FC-ASK is easy to analyze. First, it is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is **complete** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses.

## BACKWARD CHAINING

The second major family of logical inference algorithms uses the **backward chaining** approach for definite clauses. These algorithms work backward from the goal, chaining through rules to find known facts that Support the proof.

**A backward-chaining algorithm**

Figure 9.6 shows a backward-chaining algorithm for definite clauses. FOL-BC-ASK(KB, goal ) will be proved if the knowledge base contains a clause of the form lhs ⇒ goal, where lhs (left-hand side) is a list of conjuncts. An atomic fact like American(West) is considered as a clause whose lhs is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query Person(x) could be proved with

the substitution {x/John} as well as with {x/Richard }. So we implement FOL-BC-ASK as a **generator**— a function that returns multiple times, each time giving one possible result.

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ′ in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ′

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first,rest ← FIRST(goals), REST(goals)
    for each θ′ in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ″ in FOL-BC-AND(KB, rest, θ′) do
        yield θ″
```

**Figure 9.6** A simple backward-chaining algorithm for first-order knowledge bases.

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the rhs of the clause does indeed unify with the goal, proving every conjunct in the lhs, using FOL-BC-AND. That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go. Figure 9.7 is the proof tree for deriving Criminal (West) from sentences (9.3) through (9.10).

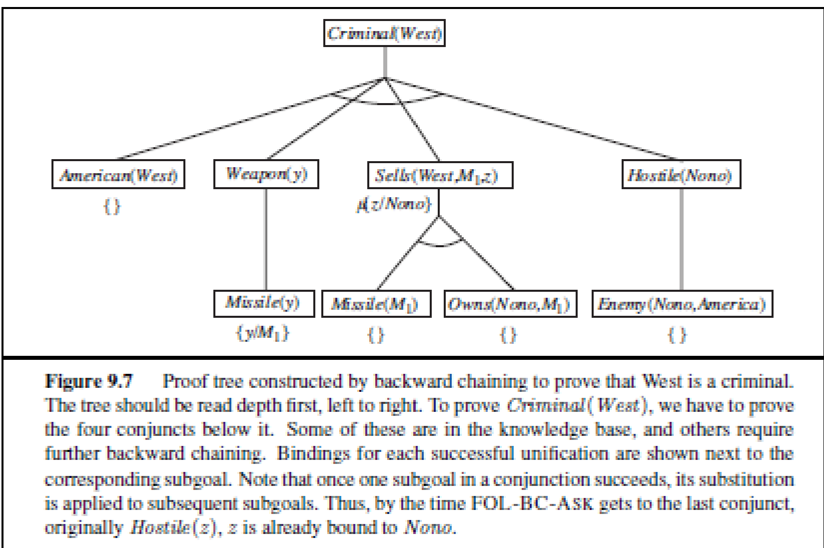



**Figure 9.7** Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal*(*West*), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally *Hostile*($z$), $z$ is already bound to *Nono*.

Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness.

**RESOLUTION**

We describe how to extend resolution to first-order logic.

**Conjunctive normal form for first-order logic**

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals.6 Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$\forall x\ American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x)$

becomes, in CNF,

$\neg American(x) \lor \neg Weapon(y) \lor \neg Sells(x, y, z) \lor \neg Hostile(z) \lor Criminal(x)$

*Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.*

We illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or

$\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y\ Loves(y, x)]$

The steps are as follows:

· **Eliminate implications**:

$\forall x\ [\neg \forall y\ \neg Animal(y) \lor Loves(x, y)] \lor [\exists y\ Loves(y, x)]$

· **Move ¬ inwards**: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$\neg \forall x\ p$ *becomes* $\exists x\ \neg p$

$\neg \exists x\ p$ *becomes* $\forall x\ \neg p$ .

Our sentence goes through the following transformations:

$\forall x\ [\exists y\ \neg(\neg Animal(y) \lor Loves(x, y))] \lor [\exists y\ Loves(y, x)]$

$\forall x\ [\exists y\ \neg\neg Animal(y) \land \neg Loves(x, y)] \lor [\exists y\ Loves(y, x)]$

$\forall x\ [\exists y\ Animal(y) \land \neg Loves(x, y)] \lor [\exists y\ Loves(y, x)]$

Notice how a universal quantifier ($\forall y$) in the premise of the **implication has** become an existential quantifier. The sentence now reads "Either there is some animal that $x$ doesn't love, or (if this is not the case) someone loves $x$." Clearly, the meaning of the original sentence has been preserved.

· **Standardize variables**: For sentences like $(\exists x P(x)) \lor (\exists x Q(x))$ which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

$\forall x\ [\exists y\ Animal(y) \land \neg Loves(x, y)] \lor [\exists z\ Loves(z, x)]$

· **Skolemize**: **Skolemization** is the process of removing existential quantifiers by elimination.

In the simple case, it is just like the Existential Instantiation rule of Section 9.1: translate $\exists x\ P(x)$ into $P(A)$, where $A$ is a new constant. However, we can't apply Existential Instantiation to our sentence above because it doesn't match the pattern $\exists v\ \alpha$; only parts of the sentence match the pattern. If we blindly apply the rule to the two matching parts we get

$\forall x\ [Animal(A) \land \neg Loves(x, A)] \lor Loves(B, x)$

which has the wrong meaning entirely: it says that everyone either fails to love a particular animal $A$ or is loved by some particular entity $B$. In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person. Thus, we want the Skolem entities to depend on $x$ and $z$:

$\forall x\ [Animal(F(x)) \land \neg Loves(x, F(x))] \lor Loves(G(z), x)$

Here F and G are **Skolem functions**. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable.

· **Drop universal quantifiers**: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers:

*[Animal (F(x)) ∧ ¬Loves(x, F(x))] ∨ Loves(G(z), x)*

· **Distribute ∨ over ∧**:

*[Animal (F(x)) ∨ Loves(G(z), x)] ∧ [¬Loves(x, F(x)) ∨ Loves(G(z), x)]*

This step may also require flattening out nested conjunctions and disjunctions.

Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one *unifies with* the negation of the other. Thus, we have

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\textsc{Subst}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where $\textsc{Unify}(\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{and} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

by eliminating the complementary literals $Loves(G(x), x)$ and $\neg Loves(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

$$[Animal(F(x)) \vee \neg Kills(G(x), x)] .$$

This rule is called the **binary resolution** rule because it resolves exactly two literals.

**EXAMPLE 1**

Consider the crime example. The sentences in CNF are

*¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x, y, z) ∨ ¬Hostile(z) ∨ Criminal (x)*

*¬Missile(x) ∨ ¬Owns(Nono, x) ∨ Sells(West, x, Nono)*

*¬Enemy(x, America) ∨ Hostile(x)*

*¬Missile(x) ∨Weapon(x)*

*Owns(Nono, M1) Missile(M1)*

*American(West) Enemy(Nono, America)*

We also include the negated goal ¬Criminal (West). The resolution proof is shown in Figure 9.11.

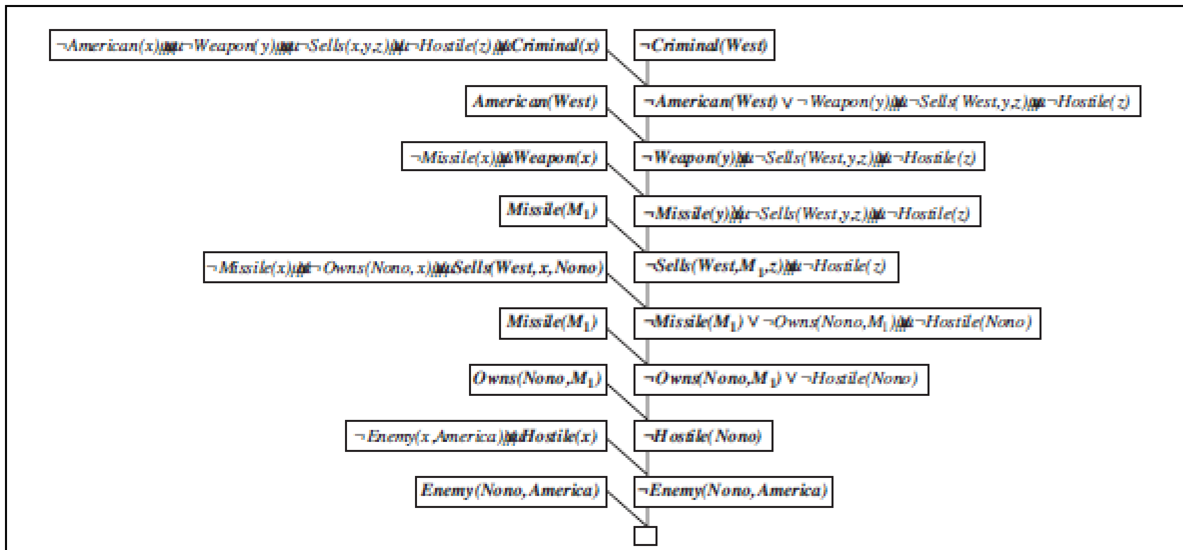



**Figure 9.11** A resolution proof that West is a criminal. At each step, the literals that unify are in bold.

Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond *exactly* to the consecutive values of the goals variable in the backward-chaining algorithm of Figure 9.6. This is because we always choose to resolve with a clause whose positive literal unified with the leftmost literal of the "current" clause on the spine; this is exactly what happens in backward chaining. Thus, backward chaining is just a special case of resolution with a particular control strategy to decide which resolution to perform next.

**EXAMPLE 2**

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is as follows:

*Everyone who loves all animals is loved by someone.*
*Anyone who kills an animal is loved by no one.*
*Jack loves all animals.*
*Either Jack or Curiosity killed the cat, who is named Tuna.*
*Did Curiosity kill the cat?*

First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall x \; [\forall y \;\; Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y \;\; Loves(y, x)]$

B. $\forall x \; [\exists z \;\; Animal(z) \land Kills(x, z)] \Rightarrow [\forall y \;\; \neg Loves(y, x)]$

C. $\forall x \;\; Animal(x) \Rightarrow Loves(Jack, x)$

D. $Kills(Jack, Tuna) \lor Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\forall x \;\; Cat(x) \Rightarrow Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

Now we apply the conversion procedure to convert each sentence to CNF:

A1. $Animal(F(x)) \lor Loves(G(x), x)$

A2. $\neg Loves(x, F(x)) \lor Loves(G(x), x)$

B. $\neg Loves(y, x) \lor \neg Animal(z) \lor \neg Kills(x, z)$

C. $\neg Animal(x) \lor Loves(Jack, x)$

D. $Kills(Jack, Tuna) \lor Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\neg Cat(x) \lor Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

The resolution proof that Curiosity killed the cat is given in Figure 9.12. In English, the proof could be paraphrased as follows:
*Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore, Curiosity killed the cat.*

**Figure 9.12** A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $Loves(G(Jack), Jack)$. Notice also in the upper right, the unification of $Loves(x, F(x))$ and $Loves(Jack, x)$ can only succeed after the variables have been standardized apart.

The proof answers the question "Did Curiosity kill the cat?" but often we want to pose more general questions, such as "Who killed the cat?" Resolution can do this, but it takes a little more work to obtain the answer. The goal is $\exists w \; Kills(w, Tuna)$, which, when negated, becomes $\neg Kills(w, Tuna)$ in CNF. Repeating the proof in Figure 9.12 with the new negated

goal, we obtain a similar proof tree, but with the substitution *{w/Curiosity}* in one of the steps. So, in this case, finding out who killed the cat is just a matter of keeping track of the bindings for the query variables in the proof.

**EXAMPLE 3**

1. All people who are graduating are happy.
2. All happy people smile.
3. Someone is graduating.
4. Conclusion: Is someone smiling?

**Solution:**

**Convert the sentences into predicate Logic**

1. ∀x graduating(x) -> happy(x)
2. ∀x happy(x) -> smile(x)
3. ∃x graduating(x)
4. ∃x smile(x)

**Convert to clausal form**

**(i) Eliminate the → sign**

1. ∀x¬graduating(x) ∨happy(x)
2. ∀x¬happy(x) ∨smile(x)
3. ∃x graduating(x)
4. ¬∃x smile(x)

**(ii) Reduce the scope of negation**

1. ∀x¬graduating(x) ∨happy(x)
2. ∀x¬happy(x) ∨smile(x)
3. ∃x graduating(x)
4. ∀x ¬ smile(x)

**(iii) Standardize variables apart**

1. ∀x¬graduating(x) ∨happy(x)
2. ∀y¬happy(y) ∨smile(y)
3. ∃x graduating(z)
4. ∀w ¬ smile(w)

**(iv) Move all quantifiers to the left**

1. ∀x¬graduating(x) ∨happy(x)
2. ∀y¬happy(y) ∨smile(y)
3. ∃x graduating(z)
4. ∀w ¬ smile(w)

**(v) Eliminate ∃**

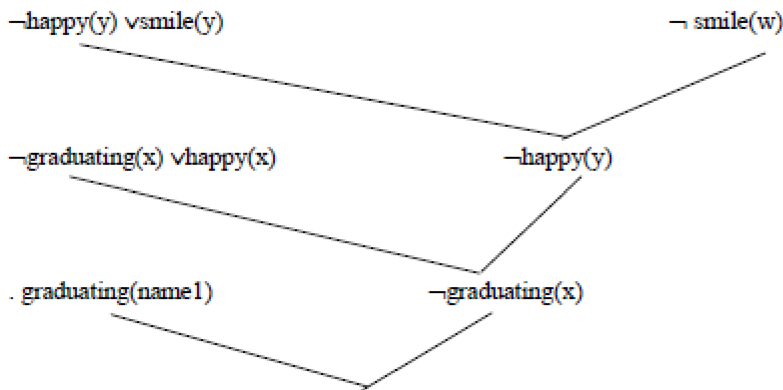
1. ∀x¬graduating(x) ∨happy(x)
2. ∀x¬happy(y) ∨smile(y)
3. graduating(name1)
4. ∀w ¬ smile(w)

**(vi) Eliminate ∀**

1. ¬graduating(x) ∨happy(x)
2. ¬happy(y) ∨smile(y)
3. graduating(name1)
4. ¬ smile(w)

**(vii) Convert to conjunct of disjuncts form.**

**(viii) Make each conjunct a separate clause.**
**(ix) Standardize variables apart again.**




**None**
**Thus, we proved someone is smiling.**

**EXAMPLE 4**
Explain the unification algorithm used for reasoning under predicate logic with an example.
Consider the following facts
a. Team India
b. Team Australia
c. Final match between India and Australia
d. India scored 350 runs, Australia scored 350 runs, India lost 5 wickets, Australia lost 7 wickets.
e. The team which scored the maximum runs wins.
f. If the scores are same the team which lost minimum wickets wins the match.
Represent the facts in predicate, convert to clause form and prove by resolution “India wins the match”.

**Solution:**

**Convert in to predicate Logic**

(a) team(India)

(b) team(Australia)

(c) team(India) ∧ team(Australia)→final_match(India, Australia)

(d) score(India,350) ∧ score(Australia,350) ∧wicket(India,5) ∧ wicket(Australia,7)

(e) ∃x team(x) ∧ wins(x)→score(x, max_runs))

(f) ∃xy score(x,equal(y)) ∧ wicket(x, min) ∧final_match(x,y)→win(x)

**Convert to clausal form**

**(i) Eliminate the → sign**

(a) team(India)

(b) team(Australia)

(c) ¬ (team(India) ∧ team(Australia)) ∨ final_match(India, Australia)

(d) score(India,350) ∧ score(Australia,350) ∧wicket(India,5) ∧ wicket(Australia,7)

(e) ∃x ¬ (team(x) ∧ wins(x)) ∨score(x, max_runs))

(f) ∃xy ¬ (score(x,equal(y)) ∧ wicket(x, min) ∧final_match(x,y)) ∨win(x)

**(ii) Reduce the scope of negation**

(a) team(India)

(b) team(Australia)

(c) ¬ team(India) ∨ ¬team(Australia) ∨ final_match(India, Australia)

(d) score(India,350) ∧ score(Australia,350) ∧wicket(India,5) ∧ wicket(Australia,7)

(e) ∃x ¬ team(x) ∨ ¬wins(x) ∨score(x, max_runs))

(f) ∃xy ¬ score(x,equal(y)) ∨¬wicket(x, min_wicket) ∨¬final_match(x,y)) ∨win(x)

**(iii) Standardize variables apart**

**(iv) Move all quantifiers to the left**

**(v) Eliminate ∃**

(a) team(India)

(b) team(Australia)

(c) ¬ team(India) ∨ ¬team(Australia) ∨ final_match(India, Australia)

(d) score(India,350) ∧ score(Australia,350) ∧wicket(India,5) ∧ wicket(Australia,7)

(e) ¬ team(x) ∨ ¬wins(x) ∨score(x, max_runs))

(f) ¬ score(x,equal(y)) ∨¬wicket(x, min_wicket) ∨¬final_match(x,y)) ∨win(x)

**(vi) Eliminate ∀**

**(vii) Convert to conjunct of disjuncts form.**

**(viii) Make each conjunct a separate clause.**

(a) team(India)

(b) team(Australia)

(c) ¬ team(India) ∨ ¬team(Australia) ∨ final_match(India, Australia)

(d) score(India,350)

score(Australia,350)

wicket(India,5)

wicket(Australia,7)

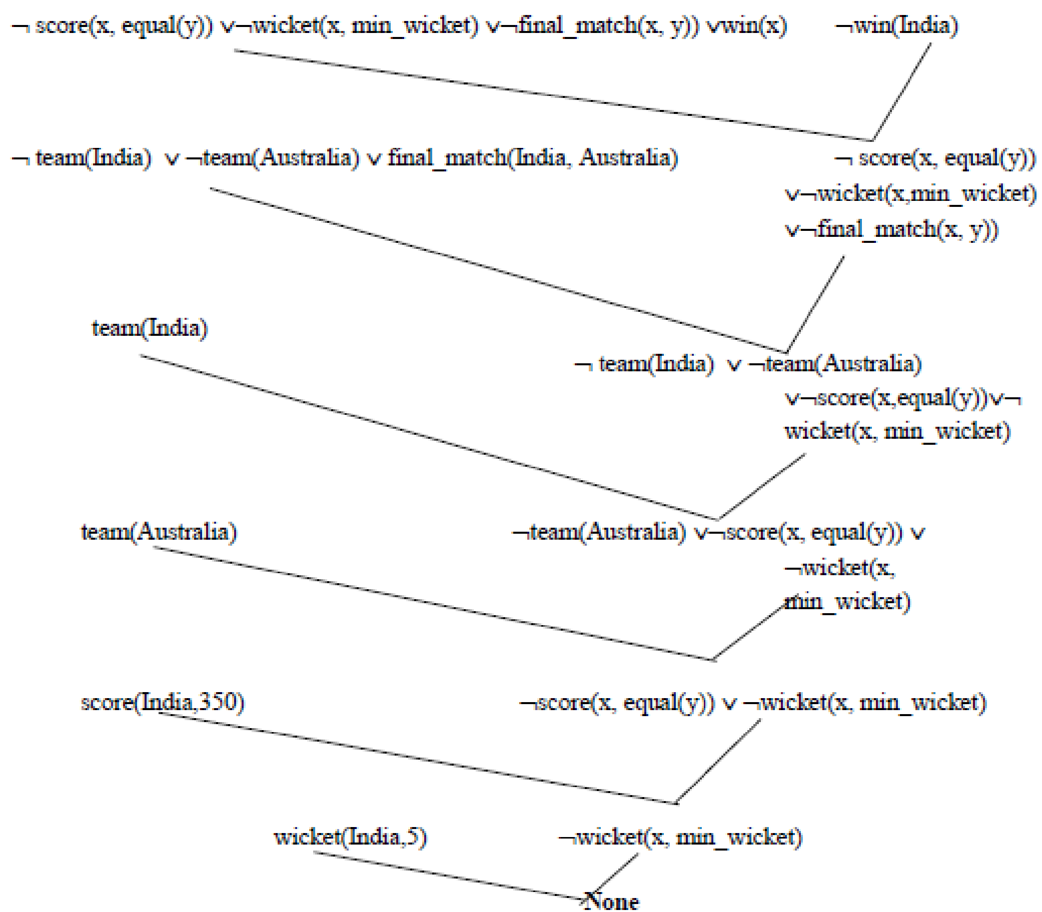
(e) ¬ team(x) ∨ ¬wins(x) ∨score(x, max_runs))

(f) ¬ score(x, equal(y)) ∨¬wicket(x, min_wicket) ∨¬final_match(x, y)) ∨win(x)

**(ix) Standardize variables apart again.**

**<u>To prove:</u>** win(India)

**<u>Disprove:</u>** ¬win(India)




**Thus, proved India wins match.**

## EXAMPLE 5

**Problem 3**

**Consider the following facts and represent them in predicate form:**

**F1. There are 500 employees in ABC company.**
**F2. Employees earning more than Rs. 5000 pay tax.**
**F3. John is a manager in ABC company.**
**F4. Manager earns Rs. 10,000.**

**Convert the facts in predicate form to clauses and then prove by resolution: "John pays tax".**

**Solution:**

**Convert in to predicate Logic**

1. company(ABC) ∧employee(500,ABC)
2. ∃x company(ABC) ∧employee(x, ABC) ∧earns(x,5000)→pays(x, tax)
3. manager(John, ABC)
4. ∃x manager(x, ABC)→earns(x,10000)

**Convert to clausal form**

**(i) Eliminate the → sign**

1. company(ABC) ∧employee(500,ABC)
2. ∃x¬ (company(ABC) ∧employee(x, ABC) ∧earns(x,5000)) ∨ pays(x, tax)
3. manager(John, ABC)
4. ∃x ¬manager(x, ABC) ∨ earns(x, 10000)

**(ii) Reduce the scope of negation**

1. company(ABC) ∧employee(500,ABC)
2. ∃x¬ company(ABC) ∨¬employee(x, ABC) ∨¬earns(x,5000) ∨ pays(x, tax)
3. manager(John, ABC)
4. ∃x ¬manager(x, ABC) ∨ earns(x, 10000)

**(iii) Standardize variables apart**

1. company(ABC) ∧employee(500,ABC)
2. ∃x¬ company(ABC) ∨¬employee(x, ABC) ∨¬earns(x,5000) ∨ pays(x, tax)
3. manager(John, ABC)
4. ∃y ¬manager(x, ABC) ∨ earns(y, 10000)

**(iv) Move all quantifiers to the left**

**(v) Eliminate ∃**

1. company(ABC) ∧employee(500,ABC)
2. ¬ company(ABC) ∨¬employee(x, ABC) ∨¬earns(x,5000) ∨ pays(x, tax)
3. manager(John, ABC)
4. ¬manager(x, ABC) ∨ earns(y, 10000)

**(vi) Eliminate ∀**

**(vii) Convert to conjunct of disjuncts form.**

**(viii) Make each conjunct a separate clause.**

1. (a)company(ABC)
   (b)employee(500,ABC)
2. ¬ company(ABC) ∨¬employee(x, ABC) ∨¬earns(x,5000) ∨ pays(x, tax)
3. manager(John, ABC)
4. ¬manager(x, ABC) ∨ earns(y, 10000)

**(ix) Standardize variables apart again.**

**Prove :**pays(John, tax)

**Disprove:** ¬pays(John, tax)

¬ company(ABC) ∨¬employee(x, ABC) ∨¬earns(x,5000) ∨ pays(x, tax)    ¬pays(John, tax)

¬manager(x, ABC) ∨ earns(y, 10000)    ¬ company(ABC) ∨¬employee(x,ABC) ∨¬earns(x,5000)

manager(John, ABC)    ¬manager(x, ABC) ∨¬company(ABC) ∨¬employee(x, ABC)

company(ABC)    ¬company(ABC) ∨¬employee(x, ABC)

employee(500,ABC)    ¬employee(x, ABC)

**None**

**Thus, proved john pays tax.**

## EXAMPLE 6 & EXAMPLE 7

**Problem 4**

If a perfect square is divisible by a prime p then it is also divisible by square of p.

Every perfect square is divisible by some prime.

36 is a perfect square.

**Convert in to predicate Logic**

1. ∀xy perfect_sq(x) ∧prime(y) ∧divides(x,y)→divides(x,square(y))
2. ∀x∃y perfect_sq(x) ∧prime(y) ∧divides(x,y)
3. perfect_sq(36)

**Problem 5**

**1. Marcus was a man**

man(Marcus)

**2. Marcus was a Pompeian**

Pompeian(Marcus)

**3. All Pompeians were Romans**

∀x (Pompeian(x)→ Roman(x))

**4. Caesar was a ruler**

ruler(Caesar)

**5. All Romans were either loyal to Caesar or hated him**

∀x (Roman(x) → loyalto(x,Caesar) ∨ hate(x,Caesar))

**6. Everyone is loyal to someone**

∀x ∃y (person(x) → person(y) ∧ loyalto(x,y))

**7. People only try to assassinate rulers they are not loyal to**

∀x ∀y (person(x) ∧ ruler(y) ∧ tryassassinate(x,y) → ¬loyalto(x,y))

**8. Marcus tried to assassinate Caesar**

tryassassinate(Marcus, Caesar)

**9. All men are persons**

∀x (man(x) → person(x))

## Difference between Predicate Logic and Propositional Logic

| Sl.No | Predicate logic | Propositional logic |
|---|---|---|
| 1. | Predicate logic is a generalization of propositional logic that allows us to express and infer arguments in infinite models. | A proposition is a declarative statement that's either TRUE or FALSE (but not both). |
| 2. | Predicate logic (also called predicate calculus and first-order logic) is an extension of propositional logic to formulas involving terms and predicates. The full predicate logic is undecidable | Propositional logic is an axiomatization of Boolean logic. Propositional logic is decidable, for example by the method of truth table |
| 3. | Predicate logic have variables | Propositional logic has variables. Parameters are all constant |
| 4. | A predicate is a logical statement that depends on one or more variables (not necessarily Boolean variables) | Propositional logic deals solely with propositions and logical connectives |
| 5. | Predicate logic there are objects, properties, functions (relations) are involved | Proposition logic is represented in terms of Boolean variables and logical connectives |
| 6. | In predicate logic, we symbolize subject and predicate separately. Logicians often use lowercase letters to symbolize subjects (or objects) and uppercase letter to symbolize predicates. | In propositional logic, we use letters to symbolize entire propositions. Propositions are statements of the form "x is y" where x is a subject and y is a predicate. |
| 7. | Predicate logic uses quantifiers such as universal quantifier ("∀"), the existential quantifier ("∃") | Prepositional logic has no quantifiers. |
| 8. | **Example**<br>Everything is green" as "∀x Green(x)"<br>or<br>"Something is blue" as "∃x Blue(x)". | **Example**<br>Everything is green" as "G(x)"<br>or<br>"Something is blue" as "B(x)". |

**KNOWLEDGE REPRESENTATION**

**ONTOLOGICAL ENGINEERING**

Concepts such as *Events, Time, Physical Objects*, and *Beliefs*— that occur in many different domains. Representing these abstract concepts is sometimes called **ontological engineering**.

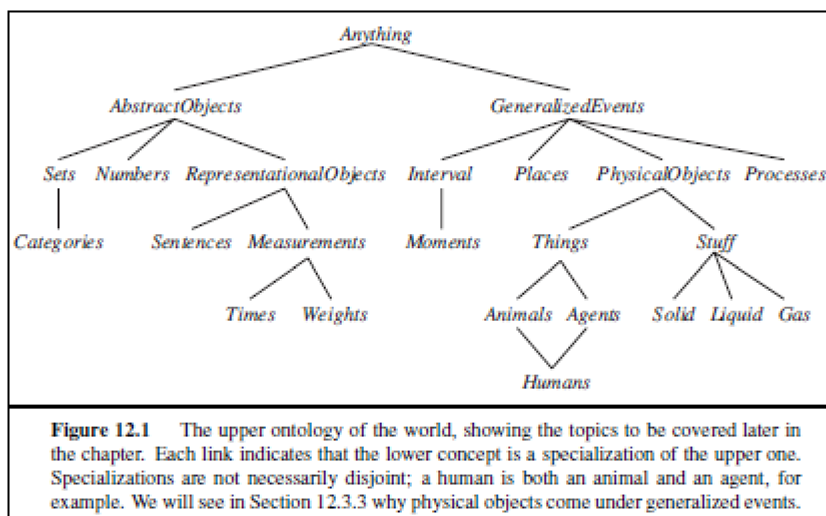



**Figure 12.1** The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint; a human is both an animal and an agent, for example. We will see in Section 12.3.3 why physical objects come under generalized events.

The general framework of concepts is called an **upper ontology** because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them, as in Figure 12.1.

**Categories and Objects**

The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, *much reasoning takes place at the level of categories.* For example, a shopper would normally have the goal of buying a basketball, rather than a *particular* basketball such as BB9

There are two choices for representing categories in first-order logic: predicates and objects. That is, we can use the predicate Basketball (b), or we can **reify**1 the category as an object, Basketballs. We could then say Member(b, Basketballs ), which we will abbreviate as $b \in Basketballs$, to say that b is a member of the category of basketballs. We say Subset(Basketballs, Balls), abbreviated as $Basketballs \subset Balls$, to say that Basketballs is a **subcategory** of Balls.

Categories serve to organize and simplify the knowledge base through **inheritance**. If we say that all instances of the category Food are edible, and if we assert that Fruit is a subclass of Food and Apples is a subclass of Fruit , then we can infer that every apple is edible. We say that the individual apples **inherit** the property of edibility, in this case from their membership in the Food category.

First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members. Here are some types of facts, with examples of each:

- An object is a member of a category.

  $BB9 \in Basketballs$

- A category is a subclass of another category.

$Basketballs \subset Balls$

- All members of a category have some properties.

$(x \in Basketballs) \Rightarrow Spherical(x)$

· Members of a category can be recognized by some properties.

$Orange(x) \wedge Round(x) \wedge Diameter(x)=9.5$ __ $\wedge\ x \in Balls \Rightarrow x \in Basketballs$

· A category as a whole has some properties.

$Dogs \in DomesticatedSpecies$

Notice that because Dogs is a category and is a member of DomesticatedSpecies , the latter must be a category of categories.

Categories can also be *defined* by providing necessary and sufficient conditions for membership. For example, a bachelor is an unmarried adult male:

$x \in Bachelors \Leftrightarrow Unmarried(x) \wedge x \in Adults \wedge x \in Males$

**Physical Composition**

We use the general PartOf relation to say that one thing is part of another. Objects can be grouped into part of hierarchies, reminiscent of the Subset hierarchy:

$PartOf(Bucharest, Romania)$

$PartOf(Romania, EasternEurope)$

$PartOf(EasternEurope, Europe)$

$PartOf(Europe, Earth)$

The PartOf relation is transitive and reflexive; that is,

$PartOf(x, y) \wedge PartOf(y, z) \Rightarrow PartOf(x, z)$

$PartOf(x, x)$

Therefore, we can conclude $PartOf(Bucharest, Earth)$.

For example, if the apples are Apple1, Apple2, and Apple3, then

$BunchOf(\{Apple1, Apple2, Apple3\})$

denotes the composite object with the three apples as parts (not elements).

We can define BunchOf in terms of the PartOf relation. Obviously, each element of s is part of BunchOf (s):

$\forall x\ x \in s \Rightarrow PartOf(x, BunchOf(s))$

Furthermore, BunchOf (s) *is the smallest object satisfying this condition*. In other words, BunchOf (s) must be part of any object that has all the elements of s as parts:

$\forall y\ [\forall x\ x \in s \Rightarrow PartOf(x, y)] \Rightarrow PartOf(BunchOf(s), y)$

**Measurements**

In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called **measures**.

$Length(L1)=Inches(1.5)=Centimeters(3.81)$

Conversion between units is done by equating multiples of one unit to another:

$Centimeters(2.54 \times d)=Inches(d)$

Similar axioms can be written for pounds and kilograms, seconds and days, and dollars and cents. Measures can be used to describe objects as follows:

$$Diameter(Basketball_{12}) = Inches(9.5)$$

$$ListPrice(Basketball_{12}) = \$(19)$$

$$d \in Days \Rightarrow Duration(d) = Hours(24)$$

**Time Intervals**

Event calculus opens us up to the possibility of talking about time, and time intervals. We will consider two kinds of time intervals: moments and extended intervals. The distinction is that only moments have zero duration:

$$Partition(\{Moments, ExtendedIntervals\}, Intervals)$$

$$i \in Moments \Leftrightarrow Duration(i) = Seconds(0)$$

The functions Begin and End pick out the earliest and latest moments in an interval, and the function Time delivers the point on the time scale for a moment. The function Duration gives the difference between the end time and the start time.

$$Interval(i) \Rightarrow Duration(i) = (Time(End(i)) - Time(Begin(i)))$$

$$Time(Begin(AD1900)) = Seconds(0)$$

$$Time(Begin(AD2001)) = Seconds(3187324800)$$

$$Time(End(AD2001)) = Seconds(3218860800)$$

$$Duration(AD2001) = Seconds(31536000)$$

Two intervals Meet if the end time of the first equals the start time of the second. The complete set of interval relations, as proposed by Allen (1983), is shown graphically in Figure 12.2 and logically below:

$$Meet(i, j) \Leftrightarrow End(i) = Begin(j)$$

$$Before(i, j) \Leftrightarrow End(i) < Begin(j)$$

$$After(j, i) \Leftrightarrow Before(i, j)$$

$$During(i, j) \Leftrightarrow Begin(j) < Begin(i) < End(i) < End(j)$$

$$Overlap(i, j) \Leftrightarrow Begin(i) < Begin(j) < End(i) < End(j)$$

$$Begins(i, j) \Leftrightarrow Begin(i) = Begin(j)$$

$$Finishes(i, j) \Leftrightarrow End(i) = End(j)$$

$$Equals(i, j) \Leftrightarrow Begin(i) = Begin(j) \wedge End(i) = End(j)$$

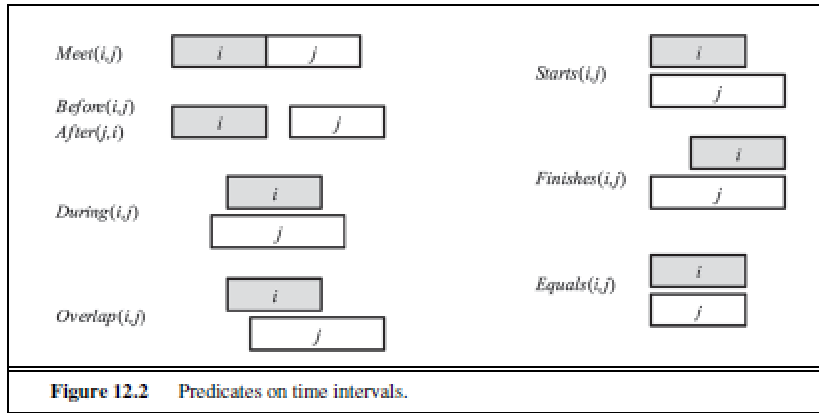


**Figure 12.2** Predicates on time intervals.

**EVENTS**

Event calculus reifies fluents and events. The fluent *At(Shankar , Berkeley)* is an object that refers to the fact of Shankar being in Berkeley, but does not by itself say anything about whether it is true. To assert that a fluent is actually true at some point in time we use the predicate T, as in *T(At(Shankar , Berkeley), t)*.

Events are described as instances of event categories. The event E1 of Shankar flying from San Francisco to Washington, D.C. is described as

$E1 \in Flyings \land Flyer(E1, Shankar) \land Origin(E1, SF) \land Destination(E1, DC)$

we can define an alternative three-argument version of the category of flying events and say

$E1 \in Flyings(Shankar, SF, DC)$

We then use Happens(E1, i) to say that the event E1 took place over the time interval i, and we say the same thing in functional form with Extent(E1)=i. We represent time intervals by a (start, end) pair of times; that is, $i = (t1, t2)$ is the time interval that starts at t1 and ends at t2. The complete set of predicates for one version of the event calculus is

*T(f, t)* Fluent f is true at time t

*Happens(e, i)* Event e happens over the time interval i

*Initiates(e, f, t)* Event e causes fluent f to start to hold at time t

*Terminates(e, f, t)* Event e causes fluent f to cease to hold at time t

*Clipped(f, i)* Fluent f ceases to be true at some point during time interval i

*Restored (f, i)* Fluent f becomes true sometime during time interval i

We assume a distinguished event, Start , that describes the initial state by saying which fluents are initiated or terminated at the start time. We define T by saying that a fluent holds at a point in time if the fluent was initiated by an event at some time in the past and was not made false (clipped) by an intervening event. A fluent does not hold if it was terminated by an event and not made true (restored) by another event. Formally, the axioms are:

$Happens(e, (t1, t2)) \land Initiates(e, f, t1) \land \neg Clipped(f, (t1, t)) \land t1 < t \Rightarrow T(f, t)$

$Happens(e, (t1, t2)) \land Terminates(e, f, t1) \land \neg Restored(f, (t1, t)) \land t1 < t \Rightarrow \neg T(f, t)$

where Clipped and Restored are defined by

$Clipped(f, (t1, t2)) \Leftrightarrow \exists\ e, t, t3\ Happens(e, (t, t3)) \wedge t1 \leq t < t2 \wedge Terminates(e, f, t)$

$Restored(f, (t1, t2)) \Leftrightarrow \exists\ e, t, t3\ Happens(e, (t, t3)) \wedge t1 \leq t < t2 \wedge Initiates(e, f, t)$

## MENTAL EVENTS AND MENTAL OBJECTS

What we need is a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model does not have to be detailed. We do not have to be able to predict how many milliseconds it will take for a particular agent to make a deduction. We will be happy just to be able to conclude that mother knows whether or not she is sitting.

We begin with the **propositional attitudes** that an agent can have toward mental objects: attitudes such as Believes, Knows, Wants, Intends, and Informs. The difficulty is that these attitudes do not behave like "normal" predicates. For example, suppose we try to assert that Lois knows that Superman can fly:

$Knows(Lois, CanFly(Superman))$

One minor issue with this is that we normally think of CanFly(Superman) as a sentence, but here it appears as a term. That issue can be patched up just be reifying CanFly(Superman); making it a fluent. A more serious problem is that, if it is true that Superman is Clark Kent, then we must conclude that Lois knows that Clark can fly:

$(Superman = Clark) \wedge Knows(Lois, CanFly(Superman)) \models Knows(Lois, CanFly(Clark))$

**Modal logic** is designed to address this problem. Regular logic is concerned with a single modality, the modality of truth, allowing us to express "*P* is true." Modal logic includes special modal operators that take sentences (rather than terms) as arguments. For example, "*A* knows *P*" is represented with the notation **K**AP, where **K** is the modal operator for knowledge. It takes two arguments, an agent (written as the subscript) and a sentence. The syntax of modal logic is the same as first-order logic, except that sentences can also be formed with modal operators.

In first-order logic a **model** contains a set of objects and an interpretation that maps each name to the appropriate object, relation, or function. In modal logic we want to be able to consider both the possibility that Superman's secret identity is Clark and that it isn't. Therefore, we will need a more complicated model, one that consists of a collection of **possible worlds** rather than just one true world. The worlds are connected in a graph by **accessibility relations**, one relation for each modal operator. We say that world w1 is accessible from world w0 with respect to the modal operator **K**A if everything in w1 is consistent with what A knows in w0, and we write this as Acc(**K**A, w0, w1). In diagrams such as Figure 12.4 we show accessibility as an arrow between possible worlds.

In general, a knowledge atom **K**AP is true in world w if and only if P is true in every world accessible from w. The truth of more complex sentences is derived by recursive application of this rule and the normal rules of first-order logic. That means that modal logic can be used to reason about nested knowledge sentences: what one agent knows about another agent's knowledge. For example, we can say that, even though Lois doesn't know whether Superman's secret identity is Clark Kent, she does know that Clark knows:

$\mathbf{K}Lois\ [\mathbf{K}Clark\ Identity(Superman, Clark) \vee \mathbf{K}Clark \neg Identity(Superman, Clark)]$

Figure 12.4 shows some possible worlds for this domain, with accessibility relations for Lois and Superman.

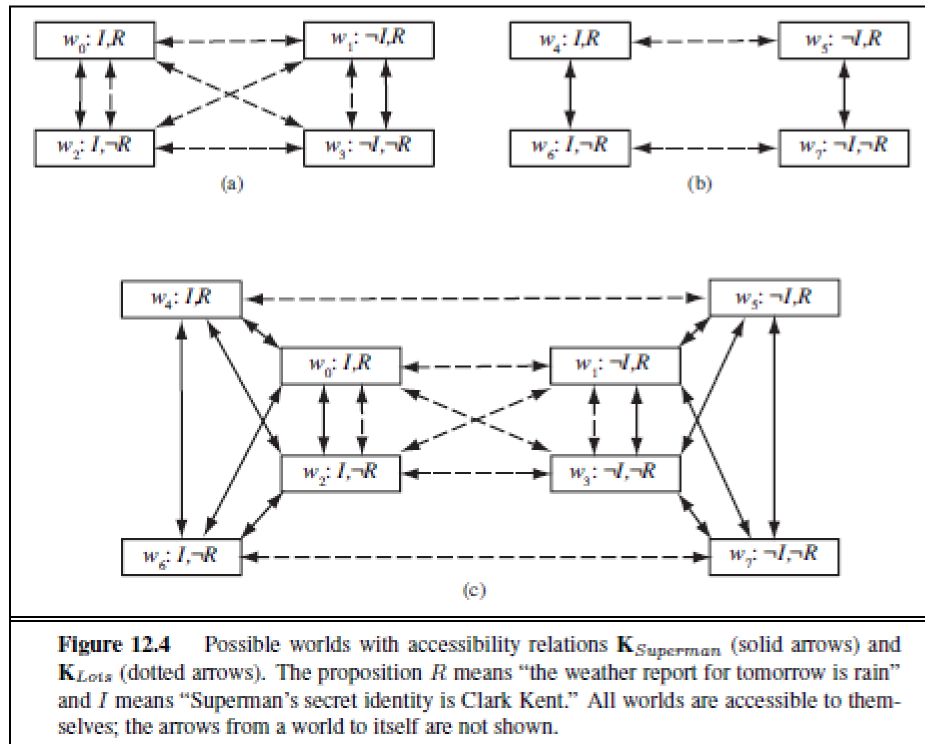



**Figure 12.4** Possible worlds with accessibility relations $\mathbf{K}_{Superman}$ (solid arrows) and $\mathbf{K}_{Lois}$ (dotted arrows). The proposition $R$ means "the weather report for tomorrow is rain" and $I$ means "Superman's secret identity is Clark Kent." All worlds are accessible to themselves; the arrows from a world to itself are not shown.

In the TOP-LEFT diagram, it is common knowledge that Superman knows his own identity, and neither he nor Lois has seen the weather report. So in w0 the worlds w0 and w2 are accessible to Superman; maybe rain is predicted, maybe not. For Lois all four worlds are accessible from each other; she doesn't know anything about the report or if Clark is Superman. But she does know that Superman knows whether he is Clark, because in every world that is accessible to Lois, either Superman knows I, or he knows ¬I. Lois does not know which is the case, but either way she knows Superman knows.

In the TOP-RIGHT diagram it is common knowledge that Lois has seen the weather report. So in w4 she knows rain is predicted and in w6 she knows rain is not predicted. Superman does not know the report, but he knows that Lois knows, because in every world that is accessible to him, either she knows R or she knows ¬R.

In the BOTTOM diagram we represent the scenario where it is common knowledge that Superman knows his identity, and Lois might or might not have seen the weather report. We represent this by combining the two top scenarios, and adding arrows to show that Superman does not know which scenario actually holds. Lois does know, so we don't need to add any arrows for her. In w0 Superman still knows I but not R, and now he does not know whether Lois knows R. From what Superman knows, he might be in w0 or w2, in which case Lois does not know whether R is true, or he could be in w4, in which case she knows R, or w6, in which case she knows ¬R.

## REASONING SYSTEMS FOR CATEGORIES

This section describes systems specially designed for organizing and reasoning with categories. There are two closely related families of systems: **semantic networks** provide graphical aids for visualizing a knowledge base and efficient algorithms for inferring properties of an object on the basis of its category membership; and **description logics** provide a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships between categories.

## SEMANTIC NETWORKS

There are many variants of semantic networks, but all are capable of representing individual objects, categories of objects, and relations among objects. A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled links. For example, Figure 12.5 has a MemberOf link between Mary and FemalePersons , corresponding to the logical assertion Mary ∈FemalePersons ; similarly, the SisterOf link between Mary and John corresponds to the assertion SisterOf (Mary, John). We can connect categories using SubsetOf links, and so on.
We know that persons have female persons as mothers, so can we draw a HasMother link from Persons to FemalePersons? The answer is no, because HasMother is a relation between a person and his or her mother, and categories do not have mothers. For this reason, we have used a special notation—the double-boxed link—in Figure 12.5. This link asserts that

$$\forall x \; x \in Persons \Rightarrow [\forall \; y \; HasMother(x, \; y) \Rightarrow y \in FemalePersons]$$

We might also want to assert that persons have two legs—that is,

$$\forall x \; x \in Persons \Rightarrow Legs(x, \; 2)$$

The semantic network notation makes it convenient to perform **inheritance** reasoning . For example, by virtue of being a person, Mary inherits the property of having two legs. Thus, to find out how many legs Mary has, the inheritance algorithm follows the MemberOf link from Mary to the category she belongs to, and then follows SubsetOf links up the hierarchy until it finds a category for which there is a boxed Legs link—in this case, the Persons category.

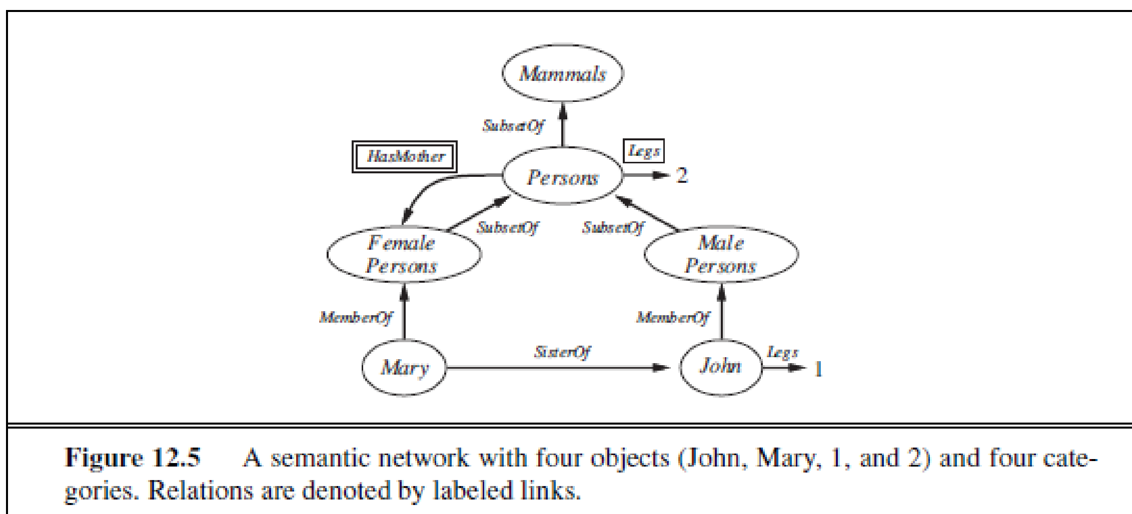



**Figure 12.5** A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

Inheritance becomes complicated when an object can belong to more than one category or when a category can be a subset of more than one other category; this is called **multiple inheritance**.
The drawback of semantic network notation, compared to first-order logic: the fact that links between bubbles represent only *binary* relations. For example, the sentence Fly(Shankar , NewYork, NewDelhi , Yesterday) cannot be asserted directly in a semantic network. Nonetheless, we *can* obtain the effect of n-ary assertions by reifying the proposition itself as an event belonging to an appropriate event category. Figure 12.6 shows the semantic network structure for this particular event. Notice that the restriction to binary relations forces the creation of a rich ontology of reified concepts.

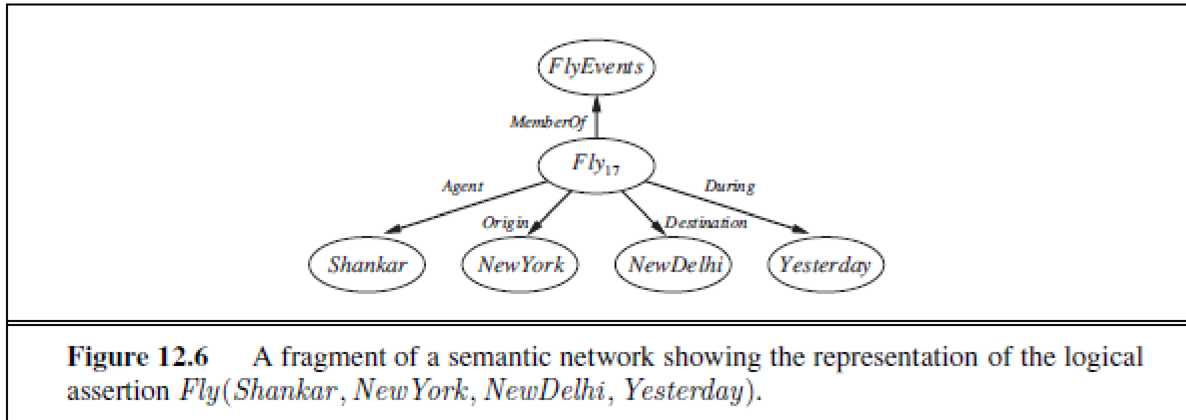



**Figure 12.6** A fragment of a semantic network showing the representation of the logical assertion $Fly(Shankar, NewYork, NewDelhi, Yesterday)$.

One of the most important aspects of semantic networks is their ability to represent **default values** for categories. Examining Figure 12.5 carefully, one notices that John has one leg, despite the fact that he is a person and all persons have two legs. In a strictly logical KB, this would be a contradiction, but in a semantic network, the assertion that all persons have two legs has only default status; that is, a person is assumed to have two legs unless this is contradicted by more specific information.