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.

3.1 FIRST-ORDER PREDICATE LOGIC

First-Order Predicate Logic (also called as Frst-Order Logic or Frst-Order Predicate Calculus, sometimes abbreviated as FOL or FOPC) is sufficiently expressive to represent a good deal of our commonsense knowledge. It also either subsumes or forms the foundation of many other representation languages.

The language of first-order logic is built around objects and relations. It has been so important to mathematics, philosophy, and artificial intelligence precisely because those fields and indeed, much of everyday human existence can be usefully thought of as dealing with objects and the relations among them. First-order logic can also express facts about some or all of the objects in the universe. This enables one to represent general laws or rules.

3.1.1 Propositional Logic

Propositional logic is a declarative, compositional semantics that is context-independent and unambiguous logic. It can be used to build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks.

- Propositional logic 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.
- Propositional logic has compositionality i.e., meaning of a sentence is a function of the meaning of its parts.

In the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to **objects** (squares, pits) and verbs and verb phrases that refer to **relations** among

objects (is breezy, is adjacent to, shoots). Some of these relations are **functions** - relations in which there is only one "value" for a given "input." It is easy to start listing examples of objects, relations, and functions:

- Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries ...
- Relations: these can be unary relations or properties such as red, round, bogus, prime, multistoried ..., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, ...
- Functions: father of, best friend, third inning of, one more than, beginning of ...

Indeed, almost any assertion can be thought of as referring to objects and properties or relations. Some examples follow:

- "One plus two equals three." Objects: one, two, three, one plus two; Relation: equals; Function: plus
- "Squares neighboring the wumpus are smelly." Objects: wumpus, squares; Property: smelly; Relation: neighboring.
- "Evil King John ruled England in 1200." Objects: John, England, 1200; Relation: ruled; Properties: evil, king.

The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language, that is, what it assumes about the nature of reality. Mathematically, this commitment is expressed through the nature of formal models with respect to which truth of sentences is defined. For example, **propositional logic** assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states: true or false, and each model assigns it to each proposition symbol. **First-order logic** assumes that the world consists of objects with certain relations among them that do or do not hold.

Special-purpose logics make still further ontological commitments; for example, **temporal logic** assumes that facts hold at particular times and that those times (which may be points or intervals) are ordered. **Higher-order logic** views the relations and functions referred to by first-order logic as objects in them. Higher-order logic is strictly more expressive than first-

order logic, in the sense that some sentences of higher-order logic cannot be expressed by any finite number of first-order logic sentences.

Logic can also be characterized by its **epistemological commitments**, the possible states of knowledge that it allows with respect to each fact. In both propositional and first order logic, a sentence represents a fact and the agent either believes the sentence to be true or false, or has no opinion. These logics therefore have three possible states of knowledge regarding any sentence. Systems using **probability theory,** on the other hand, can have any degree of belief, ranging from 0 (total disbelief) to 1 (total belief). For example, a probabilistic wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75.

Language	Ontological Commitment (What exists in the world)	EpistemologicalCommitment (What an agent believes about facts)
Propositional logic	Facts	true/false/unknown
First-order logic	Facts, objects, relations	true/false/unknown
Temporal logic	Facts, objects, relations, times	true/false/unknown
Probability theory	Facts	Degree of belief ∈ [0,1]
Fuzzy logic	Facts with degree of truth ∈ [0,1]	Known interval value

Figure 8.1: Formal languages and their ontological and epistemological commitments.

3.1.2 Syntax and Semantics of FOL

Models for first-order logic are the formal structures that constitute the possible worlds under consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. The **domain** of a model is the set of objects or domain elements it contains. The domain is required to be nonempty, i.e. 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.

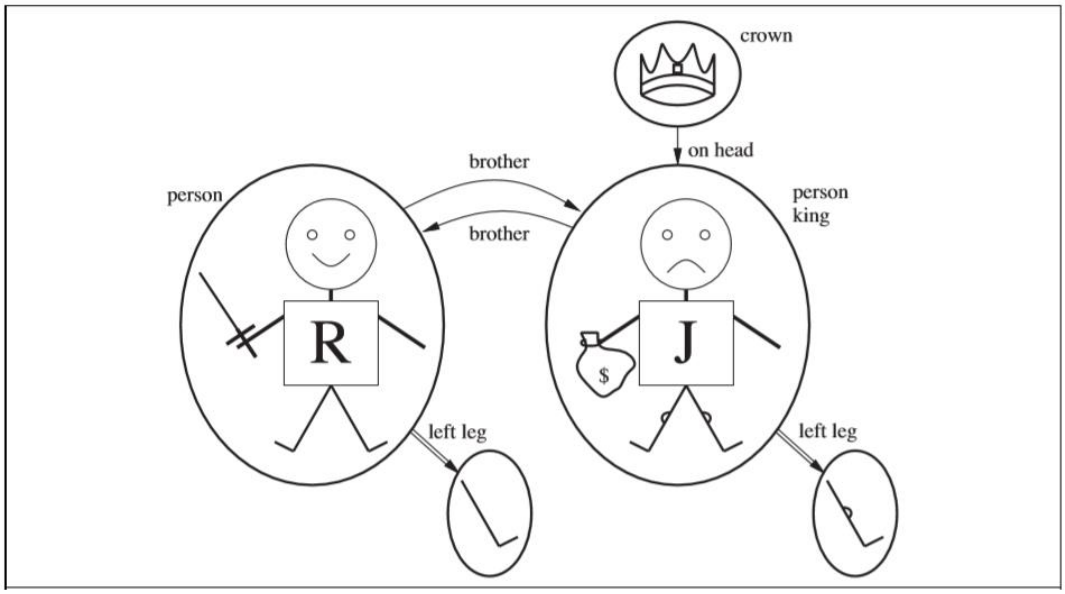


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking, a **relation** 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 > } (8.1)

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. The model also contains unary relations, or properties: the "person" property is true of both Richard and John. Strictly speaking, models in first-order logic require **total functions**, that is, there must be a value for every input tuple. For example every person must have a left leg. The basic syntactic elements of first-order logic are the symbols (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, constant symbols Richard, John; predicate symbols, OnHead, Person, King, and Crown; and the function symbol LeftLeg.

Each predicate and function symbol comes with an **arity** that fixes the number of arguments. In addition to its objects, relations, and functions, 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, which a logician would call the **intended interpretation** as follows: Richard refers to Richard the Lionheart and John refers to the evil King John. Brother refers to the brotherhood relation, 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.

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} &\rightarrow (\ \textit{Sentence}\ ) \mid [\ \textit{Sentence}\ ] \\
&\mid \neg\ \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
&\mid \textit{Quantifier}\ \textit{Variable}, \ldots\ \textit{Sentence} \\
\textit{Term} &\rightarrow \textit{Function}(\textit{Term}, \ldots) \\
&\mid \textit{Constant} \\
&\mid \textit{Variable} \\
\textit{Quantifier} &\rightarrow \forall \mid \exists \\
\textit{Constant} &\rightarrow A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\textit{Predicate} &\rightarrow \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\text{OPERATOR PRECEDENCE} &: \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form.

A **term** is a logical expression that refers 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).

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). This states, under the intended interpretation that Richard the Lionheart is the brother of King 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 (again, under a suitable interpretation).

Logical connectives can be used to construct more complex sentences, with the same syntax and semantics as in propositional calculus. For example:

- ¬ Brother(LeftLeg(Richard),John)
- Brother(Richard,John) ∧ Brother(John,Richard)
- King(Richard) ∨ King(John)
- ¬King(Richard) ⇒ King(John) .

Quantifiers 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 (∀) allows “All kings are persons,” to be written in first-order logic as

$$\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$$

∀ is usually pronounced “For all ...”. (Remember that the upside-down A stands for “all.”) Thus, the sentence says, “For all x, if x is a king, then x is a person.” The symbol x is called variable. By convention, variables are lowercase letters. A **variable** is a term all by itself can serve as argument of function, like LeftLeg(x). A term with no variables is called a **ground term**.

Intuitively, the sentence ∀x P, where P is any logical expression, says that P is true for every object x. More precisely, ∀x P is true in a given model if P is true in all **possible extended interpretations** constructed from the interpretation given in the model. The universally quantified sentence ∀x King(x) ⇒ Person(x) can extend the interpretation.

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

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

Existential quantification (∃) make statement about some object in the universe without naming it, by using an existential quantifier. For example, that King John has a crown on his head, we write ∃x Crown(x) ∧ OnHead(x,John) . ∃x is pronounced "There exists an x such that ..." or "For some x...". Intuitively, the sentence ∃x P says that P is true for at least one object x.

Nested quantifiers 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

$$\forall x \, \forall y \, \text{Brother}(x, y) \Rightarrow \text{Sibling}(x, y) .$$

The **two quantifiers are actually connected** with each other, through negation. Asserting that everyone dislikes pills is the same as asserting there does not exist someone who likes them, and vice versa.

$$\forall x \, \neg\text{Likes}(x, \text{Pills}) \text{ is equivalent to } \neg\exists x \, \text{Likes}(x,\text{Pills}).$$

$$\forall x \, \text{Likes}(x,\text{IceCream}) \text{ is equivalent to } \neg\exists x \, \neg\text{Likes}(x,\text{IceCream})$$

The **De Morgan rules** for quantified and unquantified sentences are as follows:

$\forall x \, \neg P \equiv \neg\exists x P$ $\quad \neg\forall x P \equiv \exists x \, \neg P$ $\quad \exists x P \equiv \neg\forall x \, \neg P$ $\quad \forall x P \equiv \neg\exists x \, \neg P$

$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$ $\quad \neg(P \wedge Q) \equiv \neg P \vee \neg Q$ $\quad P \wedge Q \equiv \neg(\neg P \vee \neg Q)$ $\quad P \vee Q \equiv \neg(\neg P \wedge \neg Q)$.

We can use the **equality symbol** to signify that two terms refer to the same object. For example, "Father of John is Henry" is

$$\text{Father (John)}=\text{Henry}.$$

Richard has two brothers, John and Geoffrey can be translated as,

$$\exists x,y \, \text{Brother}(x,\text{Richard}) \wedge \text{Brother}(y,\text{Richard}) \wedge \neg (x=y) \text{ or}$$

$$\text{Brother(John,Richard)} \wedge \text{Brother(Geoffrey,Richard)} \wedge \text{John} \neq \text{Geoffrey or}$$

$$\wedge\forall x \, \text{Brother}(x,\text{Richard}) \Rightarrow (x=\text{John} \vee x=\text{Geoffrey})$$

One proposal that is very popular in database systems, **Database Semantics**, works as follows. First, we insist that every constant symbol refer to a distinct object, called **unique-names assumption**. Second, we assume that atomic sentences not known to be true are in fact false, called **closed-world assumption**. Finally, we invoke domain closure, meaning that each model contains no more domain elements than those named by the constant symbols.

3.1.3 Using First Order Logic

In knowledge representation, a **domain** is just some part of the world about which we wish to express some knowledge. 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:

$$\text{TELL(KB, King(John))}$$

$$\text{TELL(KB, Person(Richard))}$$

$$\text{TELL(KB, } \forall x \text{ King}(x) \Rightarrow \text{Person}(x)) \text{ .}$$

We can ask questions of the knowledge base using ASK. For example, ASK(KB, King(John)) returns true. ASKVARS() return the value of x. Questions asked with ASK are called **queries or goals.** ASKVARS(KB,Person(x)) 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 domain of family relationships is called **kinship**. Our **kinship axioms** are also **definitions;** they have the form $\forall x,y\ P(x,y) \Leftrightarrow$ .... The axioms define the Mother function and the Husband, Male, Parent, Grandparent, and Sibling predicates in terms of other predicates. 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: $\forall m,c\ \text{Mother}(c)=m \Leftrightarrow \text{Female}(m) \wedge \text{Parent}(m,c)$.

One's husband is one's male spouse: $\forall w,h\ \text{Husband}(h,w) \Leftrightarrow \text{Male}(h) \wedge \text{Spouse}(h,w)$.

Male and female are disjoint categories: $\forall x\ \text{Male}(x) \Leftrightarrow \neg\text{Female}(x)$.

Parent and child are inverse relations: $\forall p,c\ \text{Parent}(p,c) \Leftrightarrow \text{Child}(c,p)$.

Siblinghood is symmetric: $\forall x,y\ \text{Sibling}(x,y) \Leftrightarrow \text{Sibling}(y,x)$ is a **theorem.**

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. A predicate NatNum will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S (successor). The **Peano axioms** define natural numbers and addition. Natural numbers are defined recursively:

$$\text{NatNum}(0)$$

$$\forall n\ \text{NatNum}(n) \Rightarrow \text{NatNum}(S(n)) \text{ .}$$

That is, 0 is a natural number, and for every object n, if n is a natural number, then S(n) is a natural number. So the natural numbers are 0, S(0), S(S(0)), and so on. We also need axioms to constrain the successor function:

$$\forall n\ 0 \neq S(n)\ .$$

$$\forall m,n\ m \neq n \Rightarrow S(m) \neq S(n)\ .$$

Now we can define addition in terms of the successor function:

$$\forall m\ NatNum(m) \Rightarrow +(0,m)=m$$

$$\forall m,n\ NatNum(m) \wedge NatNum(n) \Rightarrow +(S(m),n)=S(+(m,n))\ .$$

The first of these axioms says that adding 0 to any natural number m gives m itself. Second one says addition of two natural numbers. The domain of **sets** is also fundamental to mathematics as well as to common sense reasoning. We want to be able to represent individual sets, including the empty set. Build up sets by adding an element to a set or taking the union or intersection of two sets. Member function finds whether an element is a member of a set and distinguishes sets from objects that are not sets. **Lists** are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list. Nil is the constant list with no elements; Cons, Append, First, and Rest are functions; and Find is the predicate that checks members for lists.

3.1.4 The Knowledge-Engineering Process

1. Identify the task.
2. Assemble the relevant knowledge.
3. Decide on a vocabulary of predicates, functions, and constants.
4. Encode general knowledge about the domain.
5. Encode a description of the specific problem instance.
6. Pose queries to the inference procedure and get answers.
7. Debug the knowledge base.

3.2 PROLOG PROGRAMMING

Prolog (which stands for “Programming in Logic”) is one of the most interesting languages in use today. Prolog is a declarative language that focuses on logic. Programming in Prolog involves specifying rules and facts and allowing Prolog to derive a solution. This differs from typical declarative programming, where a solution is coded.

Building a Prolog application typically involves describing the landscape of the problem (using rules and facts), and then using goals to invoke execution. Prolog was created around 1972 by Alain Colmerauer and Robert Kowalski as a competitor to the LISP language.

- The earliest Prolog applications were in the domain of natural-language programming, as this was the object of research by its creators. Natural language programming (NLP) is supported by a unique feature of Prolog.
- Prolog includes a built-in mechanism for parsing context-free grammars, making it ideal for NLP or parsing grammars.
- In fact, like LISP, a Prolog interpreter can be easily implemented in Prolog.
- Today, Prolog is used primarily by researchers, but the language has advanced to support multiple programming paradigms.
- Prolog is ideal for developing knowledge-based systems such as expert systems and also systems for research into computer algebra.

3.2.1 Overview of the Prolog Language

Prolog comprised of two fundamental elements, the **database** and the **interpreter.** The database contains the facts and rules that are used to describe the problem. The interpreter provides the control in the form of a deduction method. Prolog doesn't include data types, per se, but has lexical elements. Some of the important lexical elements are atoms, numbers, variables, and lists. An **atom** is a string made up of characters (lower and upper case), digits, and the underscore. Numbers are simply sequences of digits with an optional preceding minus sign. Variables have the same format as atoms, except that the first character is always capitalized. A **list** is represented as a comma delimited set of elements, surrounded by brackets. For example

person Atom,	f_451 Atom,	'A complex atom.' Atom,
86 Number,	-451 Number,	Person Variable,
A_variable Variable,	[a, simple, list] List,	[a, [list of lists]] List.

Constructing a list is performed simply as follows:

```
| ?- [a, b, c, d] = X
X = [a,b,c,d]
Yes | ?-
```

The X variable refers to the list [a, b, c, d], which is identified by the Prolog interpreter. We can also concatenate two lists using the **append** predicate.

```
| ?- append( [[a, b], [c, d]], [[e, f]], Y ).
Y = [[a,b],[c,d],[e,f]]
```

yes | ?-

List has **length and member** functions and **[Head | Tail]** variables.

What separates Prolog from all other languages is its **built-in ability of deduction.** Define fruit(pear). Query the Prolog database, such as:

| ?- fruit(pear).

Yes | ?-

for which Prolog would reply with 'yes' (a pear is a fruit). Create a **rule** oddity to define that the tomato is both a fruit and a vegetable.

```
oddity(X) :-
fruit(X),
vegetable(X).
```

Similarly, family tree is constructed with the following rules. To be a husband, a person of the male gender is married to a person of the female gender and a wife is a female person that is also married to a male person.

```
husband(Man, Woman) :
male(Man), married(Man, Woman).
wife(Woman, Man) :
female(Woman), married(Man, Woman).
```

We create a son rule, which defines that a son is a male child of a parent. Similarly, a daughter is a female child of a parent.

```
son(Child, Parent) :
male(Child), child(Child, Parent).
daughter(Child, Parent) :
female(Child), child(Child, Parent).
```

3.2.2 Querying the family tree database with gprolog.

The prolog code to answer the query whether bud is the grand parent of marc is given below:

```
| ?- consult('family.pl').
compiling /home/mtj/family.pl for byte code... /home/mtj/family.pl
```

compiled, 71 lines

read - 5920 bytes written, 51 ms (10 ms)

yes

| ?- grandchild(marc, X).

X = bud

X = ernie

X = celeta

X = lila

No

| ?-

| ?- trace.

The debugger will first creep -- showing everything (trace)

yes

{trace}

| ?- grandparent(bud, marc).

1 1 Call: grandparent(bud,marc)

2 2 Call: parent(bud,_79)

3 3 Call: father(bud,_103)

3 3 Exit: father(bud,tim)

2 2 Exit: parent(bud,tim)

3 2 Call: parent(tim,marc)

4 3 Call: father(tim,marc)

5 3 Exit: father(tim,marc)

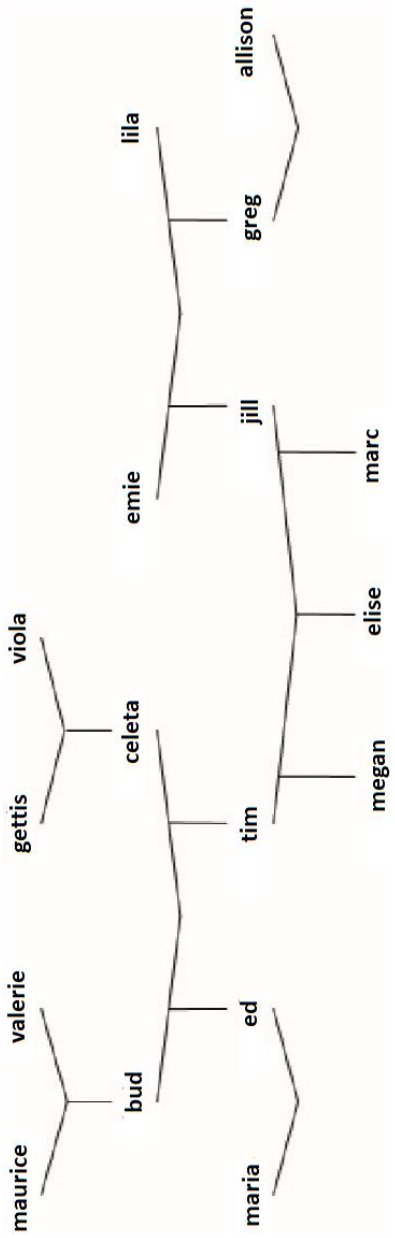
4 2 Exit: parent(tim,marc)

1 1 Exit: grandparent(bud,marc)

true ?

yes

{trace} | ?-



Let's now have a look at this database in action. This is illustrated below using the GNU prolog interpreter (gprolog). **Trace** allows us to trace the

flow of the application of rules for a given query. For example, the following query checks to see if a person is a grandparent of another person. Note here that the variables preceded by underscores are temporary variables created by Prolog during its rule checking.

3.2.3 Arithmetic Expressions

Finally, let's look at an example that combines arithmetic expressions with logic programming. In this example, we'll maintain facts about the prices of products, and then a rule that can compute the total cost of a product given a quantity (again, encoded as a fact). First, let's define our initial set of facts that maintain the price list of the available products.

cost(banana, 0.35). cost(apple, 0.25). cost(orange, 0.45). cost(mango, 1.50).

We can also provide a quantity that we plan to purchase, using another relation for the fruit, for example: qty(mango, 4).

We can then add a rule that calculates the total cost of an item purchase:

```
total_cost(X,Y) :
cost(X, Cost),
qty(X, Quantity),
Y is Cost * Quantity.
```

What this rule says is that if we have a product with a cost fact, and a quantity fact, then we can create a relation to the product cost and quantity values for the item. We can invoke this from the Prolog interpreter as: | ?- total_cost(mango, TC).

```
TC = 6.0
yes | ?-
```

This tells us that the total cost for four mangos is $6.00.

3.3 UNIFICATION

We have noticed that the propositional logic approach is rather inefficient. For example, given the query Evil(x) and the knowledge base, it seems perverse to generate sentences such as King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard).

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

$$King(John)$$

$$Greedy(John)$$

Indeed, the inference of Evil (John) from the sentences seems completely obvious. We now show how to make it completely obvious to a computer.

3.3.1 A first-order inference rule

The inference that John is evil, that is, that {x/John} solves the query Evil(x) works like this: to use the rule that greedy kings are evil, find some x such that x is a king and x is greedy, and then infer that this x is evil. More generally, if there is some substitution θ that makes each of the conjuncts of the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying θ. In this case, the substitution θ ={x/John} achieves that aim. Suppose that instead of knowing Greedy(John), we know that *everyone* is greedy:

$$\forall y \text{ Greedy}(y) .$$

Then we would still be able to conclude that Evil(John), substitute {x/John, y/John} to the implication premises King(x) and Greedy(x) and the knowledge-base sentences King(John) and Greedy(y) will make them identical. Thus, we can infer the conclusion of the implication. This inference process can be captured as **Generalized Modus Ponens.** For atomic sentences p_i, p_i', and q, where there is a substitution θ such that SUBST(θ , p_i')= SUBST(θ , pi), for all i,

$$\frac{p_1', \quad p_2', \quad \ldots, \quad p_n', \quad (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

There are n+1 premises to this rule: the n atomic sentences p_i' and the one implication. The conclusion is the result of applying the substitution θ to the consequent q. For our example:

p_1' is King(John) p1 is King(x)
p_2' is Greedy(y) p2 is Greedy(x)
θ is {x/John, y/John} q is Evil(x)

SUBST(θ , q) is Evil(John) .

For any substitution θ , p |= SUBST(θ , p)

holds by Universal Instantiation. It holds in particular for a θ that satisfies the conditions of the Generalized Modus Ponens rule. Thus, from $p_1', \ldots, p_n'$ we can infer

$$\text{SUBST}(\theta, p_1') \wedge \ldots \wedge \text{SUBST}(\theta, p_n')$$

and from the implication p1 ∧ . . . ∧ pn ⇒ q we can infer

$$\text{SUBST}(\theta, p_1) \wedge \ldots \wedge \text{SUBST}(\theta, p_n) \Rightarrow \text{SUBST}(\theta, q) .$$

Now, θ in Generalized Modus Ponens is defined so that SUBST(θ , p_i')= SUBST(θ , p_i), for all i; therefore the first of these two sentences matches the premise of the second exactly.

Hence, SUBST(θ , q) follows by Modus Ponens.

Generalized Modus Ponens is a **lifted** version of Modus Ponens since it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic.

3.3.2 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)= θ where SUBST(θ , p)= SUBST(θ , q) .

Let us look at some examples of how UNIFY should behave. Suppose we have a query Ask(Knows(John, x)): whom does John know? Answers to this query 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} .

There is one more complication: we said that 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 *moregeneral* than the second, because it

places fewer restrictions on the values of the variables. It turns out that, for every unifiable pair of expressions, there is a single **most general unifier** (MGU) that is unique up to renaming and substitution of variables.

```
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 process in Figure 9.1 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 **occur check**, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed. For example, S(x) can't unify with S(S(x)). Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

Procedure for UNIFY (L1, L2)

1. if L1 or L2 are both variables or constant then
    a. if L1 or L2 are identical then return NIL

b. Else if L1 is a variable then if L1 occurs in L2 then return Fail else return (L2/L1)
c. Else if L2 is a variable then if L2 occurs in L1 then return Fail else return (L1/L2)
d. Else return Fail.

2. If initial predicate of L1 and L2 are not equal then return Fail.
3. If length (L1) is not equal to length (L2) then return Fail.
4. Set SUBST to NIL (at the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2).
5. For i = 1 to number of elements in L1 then
    a. Call UNIFY with the i th element of L1 and i th element of L2, putting the result in S
    b. If S = Fail then return Fail
    c. If S is not equal to NIL then do
        i. Apply S to the remainder of both L1 and L2
        ii. SUBST := APPEND (S, SUBST) return SUBST.

3.3.3 Storage and retrieval

Underlying the TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some sentence in the knowledge base. The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a process is inefficient, but it works, and it's all you need to understand.

We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have some chance of unifying. For example, there is no point in trying to unify Knows(John, x) with Brother (Richard, John). We can avoid such unifications by **indexing** the facts in the knowledge base. A simple scheme called **predicate indexing** puts all the Knows facts in one bucket and all the Brother facts in another. The buckets can be stored in a hash table for efficient access.

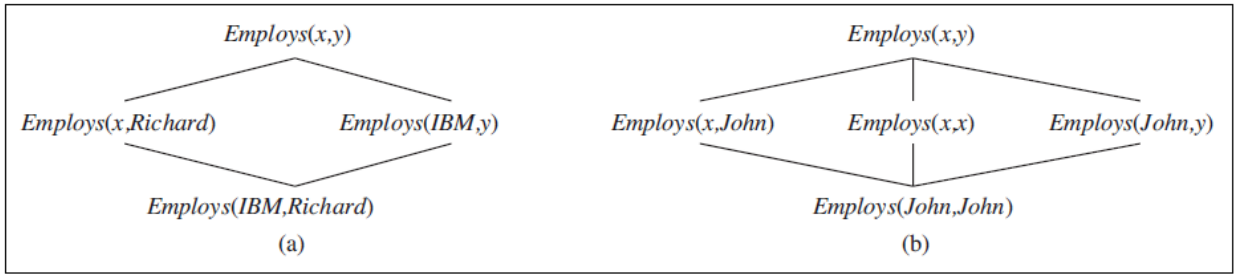


Figure 9.2 (a) The subsumption lattice whose lowest node is Employs(IBM,Richard). (b) The subsumption lattice for the sentence Employs (John, John).

Answering a query such as Employs(x,Richard) with predicate indexing would require scanning the entire bucket. For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key. Then we could simply construct the key from the query and retrieve exactly those facts that unify with the query. Therefore, facts can be stored under multiple index keys, rendering them instantly accessible to various queries that they might unify with. These queries form a **subsumption lattice**, as shown in Figure 9.2(a).

The **lattice** has some interesting properties. For example, the child of any node in the lattice is obtained from its parent by a single substitution; and the "highest" common descendant of any two nodes is the result of applying their most general unifier. The portion of the lattice above any ground fact can be constructed systematically. However for a predicate with n arguments, the lattice contains $O(2^n)$ nodes. If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored.

3.4 FORWARD CHAINING

The idea of forward-chaining algorithm is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made.

3.4.1 First-order definite clauses

Definite clauses are like Situation ⇒ Response which are especially useful for systems that make inferences in response to newly arrived information. **Definite clauses or Horn Clauses** are disjunctions of literals of which *exactly one is positive*. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent .000000000 is a single positive literal. The following are first-order definite clauses:

King(x) ∧ Greedy(x) ⇒ Evil(x) .

King(John) .

Greedy(y) .

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.

Prove : West is a criminal.

Given:

First, we will represent these facts as first-order definite clauses. ". . . it is a crime for an American to sell weapons to hostile nations":

$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$ ----- *(3.1)*

"Nono . . . has some missiles." is $\exists$ x Owns(Nono, x)$\wedge$Missile(x) which is transformed into two definite clauses by Existential Instantiation, introducing a new constant M1:

$Owns(Nono,M1)$ -------*(3.2)*

$Missile(M1)$ -------*(3.3)*

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

$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$ -------*(3.4)*

We will also need to know that missiles are weapons:

$Missile(x) \Rightarrow Weapon(x)$ -------*(3.5)*

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

$Enemy(x,America) \Rightarrow Hostile(x)$ -------*(3.6)*

"West, who is American . . .":

American(West) -------*(3.7)*

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

Enemy(Nono,America) -------*(3.8)*

3.4.2 A simple Forward-Chaining Algorithm

Starting from the known facts, trigger all the rules whose premises are satisfied, add their conclusions to the known facts. The process repeats until the query is answered or no new facts are added. Notice that a fact is not "new" if it is just a **renaming** of a known fact. One sentence is a renaming of another if they are identical except for the names of the variables. For example, Likes(x, IceCream) and Likes(y, IceCream) are renamings of each other because they differ only in the choice of x or y; their meanings are identical: everyone likes ice cream.

To Prove:

By Forward Chaining prove West is Criminal, i.e *Criminal (West).*

Solution:

Substitute { x | Nano } in equation 3.6

Enemy(Nono,America) ⇒ Hostile(Nono) -------*(3.9)*

Apply Modus Ponens in equation 3.8 and 3.9

Hostile(Nono) -------*(3.10)*

Substitute { x | M1 } in equation 3.5

Missile(M1) ⇒ Weapon(M1) -------*(3.11)*

Apply Modus Ponens to equations 3.3 and 3.11

Weapon(M1) -------*(3.12)*

Apply And Inclusion to equations 3.2 and 3.3

Missile(M1) Λ Owns (Nono, M1) -------*(3.13)*

Substitute { x | M1 } in equation 3.4

Missile(M1) ⋀ Owns(Nono, M1) ⇒ Sells(West, M1, Nono) -------*(3.14)*

Apply Modus Ponens to equations 3.13 and 3.14

Sells(West, M1, Nono) -------*(3.15)*

Apply And Inclusion to equations 3.7, 3.10, 3.12 and 3.15

American(West) Λ Weapon(M1) Λ Sells(West, M1, Nono) Λ Hostile(Nono) -------*(3.16)*

Substitute { x | West, y | M1, z | Nono } in equation 3.1

American(West) Λ Weapon(M1) Λ Sells(West, M1, Nono) Λ Hostile(Nono)

⇒ Criminal (West) ----- *(3.17)*

Apply Modus Ponens to equations 3.16 and 3.17

Criminal (West) -----*(3.18)*

Hence proved.

```
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 : Forward Chaining Algorithm

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.

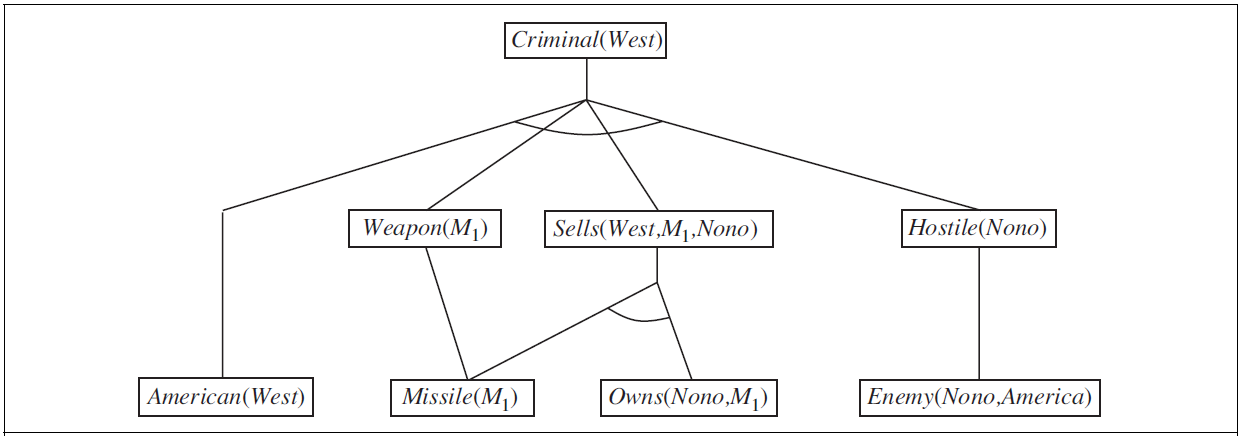


Figure 9.4 : Proof Tree of forward-chaining algorithm

3.4.3 Efficient forward chaining

The forward-chaining algorithm is easy to understand but inefficient. There are three possible sources of inefficiency.

- First, the "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive.
- Second, the algorithm **rechecks every rule on every iteration** to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration.
- Finally, the algorithm might generate many facts that are irrelevant to the goal.

Pattern Matching :

If the knowledge base contains many objects owned by Nono and very few missiles, however, it would be better to find all the missiles first and then check whether they are owned by Nono. This is the **conjunct ordering problem**, that is find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized.

Data complexity is the complexity of inference as a function of the number of ground facts in the knowledge base. The data complexity of forward chaining is polynomial. If the constraint graph forms a tree, then the CSP can be solved in linear time. Eliminate redundant rule-matching attempts in the forward-chaining algorithm.

Incremental forward chaining:

The rule-matching step fixes p_i' to match with p_i, but allows the other conjuncts of the rule to match with facts from any previous iteration. With suitable indexing, it is easy to identify all the rules that can be triggered by any given fact.

Typically, only a small fraction of the rules in the knowledge base are actually triggered by the addition of a given fact. This means that a great deal of redundant work is done in repeatedly constructing partial matches that have some unsatisfied premises. The **rete** algorithm solves this problem by binding variable flow through the network and are filtered out when they fail to match a literal.

Irrelevant facts:

The final source of inefficiency in forward chaining can be solved by, Backward chaining, restricting forward chaining to a selected subset of rules or by rewriting the rule set, using information from the goal, so that only relevant variable bindings to a so-called **magic set,** considered during forward inference.

3.5 BACKWARD CHAINING

Backward chaining approach is an inference procedure that works backward from the goal, chaining through rules to find known facts that support the proof.

- The algorithm is called with a list of goals containing a single element, the original query and returns the set of all substitutions satisfying the query.
- The goals can be thought of as a "stack". The algorithm takes the first goal in the stack and finds every clause in the knowledge base whose positive literal unifies the goal.
- Each clause creates a new recursive call in which the premise or body of the clause is added to the goal state.
- When the goal unifies with the known fact, no new sub goals are added to the stack and the goal is solved.
- The algorithm uses composition of substitutions. Compose(θ_1, θ_2) is the substitution whose effect is identical to the effect of applying each substitution in turn.

i.e, SUBST(COMPOSE(θ_1, θ_2), P) = SUBST (θ_2, SUBST(θ_1, P))

Backward chaining is a depth-first search algorithm. This means that its space requirements are linear in the size of the proof. It also means that backward chaining suffers from problems with repeated states and incompleteness.

```
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 9.7: Backward Chaining Algorithm

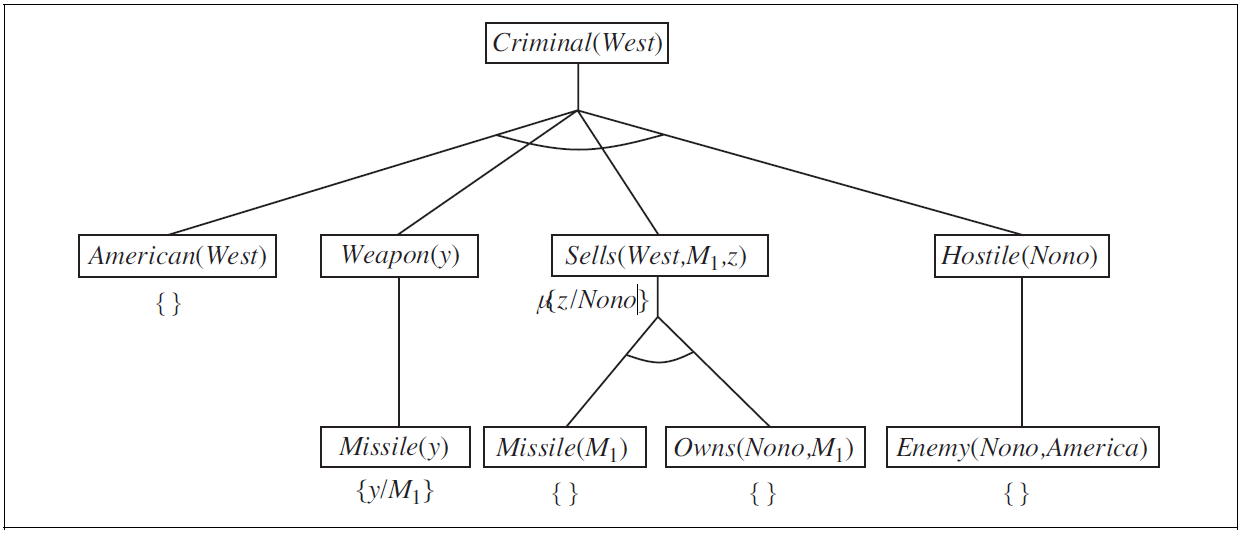


Figure 9.8 : Proof Tree of Backward Chaining Algorithm

The proof 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.

3.5.2 **Logic programming**

Prolog is the most widely used logic programming language. 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 9.8 is the proof tree for deriving Criminal (West) from sentences (3.1) through (3.8).

3.5.3 Redundant inference and infinite loops

Forward chaining on graph search problems is an example of **dynamic programming**, in which the solutions to subproblems are constructed incrementally from those of smaller subproblems and are cached to avoid recomputation. We can obtain a similar effect in a

backward chaining system using **memoization**—that is, caching solutions to subgoals as they are found and then reusing those solutions when the subgoal recurs, rather than repeating the previous computation. This is the approach taken by tabled logic programming systems, which use efficient storage and retrieval mechanisms to perform memoization. **Tabled logic programming** combines the goal-directedness of backward chaining with the dynamic programming efficiency of forward chaining.

3.5.4 Database semantics of Prolog

Database Semantics, works as follows. First, we insist that every constant symbol refer to a distinct object, called **unique-names assumption**. Second, we assume that atomic sentences not known to be true are in fact false, called **closed-world assumption**. Finally, we invoke domain closure, meaning that each model contains no more domain elements than those named by the constant symbols. There is no way to assert that a sentence is false in Prolog. This makes Prolog less expressive than first-order logic, but it is part of what makes Prolog more efficient and more concise.

3.5.5 Constraint logic programming

Constraint logic programming (CLP) allows variables to be *constrained* rather than *bound*. A CLP solution is the most specific set of constraints on the query variables that can be derived from the knowledge base. For example, the solution to the triangle(3,4,Z) query is the constraint 7 >= Z >= 1.

3.6 RESOLUTION

Conjunctive normal form for first-order logic

As in the propositional calculus, 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. 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.

The steps are as follows:

1. **Eliminate implications**:

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

2. **Move $\neg$ inwards**: 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:

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

3. **Standardize variables**: For sentences like $(\exists xP(x))\vee(\exists xQ(x))$ 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

$$\forall x\ [\exists y\ Animal\ (y) \wedge \neg Loves(x, y)]\ \vee [\exists z\ Loves(z, x)]\ .$$

4. **Skolemize**: **Skolemization** is the process of removing existential quantifiers by elimination. The Skolem entities depend on x and z: Here F and G are **Skolem functions**.

$$\forall x\ [Animal\ (F(x)) \wedge \neg Loves(x, F(x))]\ \vee Loves(G(z), x)\ .$$

5. **Drop universal quantifiers**: At this point, all remaining variables must be universally quantified.

$$[Animal\ (F(x)) \wedge \neg Loves(x, F(x))]\ \vee Loves(G(z), x)$$

6. **Distribute $\vee$ over $\wedge$**: This step may also require flattening out nested conjunctions and disjunctions.

$$[Animal\ (F(x))\ \vee Loves(G(z), x)] \wedge [\neg Loves(x, F(x))\ \vee Loves(G(z), x)]\ .$$

The resolution inference rule

For example, we can resolve the two clauses

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

by eliminating the complementary literals Loves(G(x), x) and $\neg$Loves(u, v), with unifier θ={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. The binary resolution rule by itself does not yield a complete inference procedure. The full resolution rule resolves subsets of literals in each clause that are unifiable. An alternative approach is to extend **factoring**—the removal of redundant literals—to the first-order case.

Example proofs

Resolution proves that KB |= α by proving KB ∧ ¬α unsatisfiable, that is, by deriving the empty clause. We give two example proofs. The first is the crime example from Section 9.3. The sentences in CNF are

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

$$\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono)$$

$$\neg Enemy(x,America) \vee Hostile(x)$$

$$\neg Missile(x) \vee Weapon(x)$$

$$Owns(Nono,M1)\ Missile(M1)$$

$$American(West)\ Enemy(Nono,America)\ .$$

We also include the negated goal $\neg Criminal(West)$. The resolution proof is shown in Figure 9.11. Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated.

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. ∀ x [∀ y Animal (y) ⇒ Loves(x, y)] ⇒ [∃ y Loves(y, x)]

B. ∀ x [∃ z Animal (z) ∧ Kills(x, z)] ⇒ [∀ y ¬Loves(y, x)]

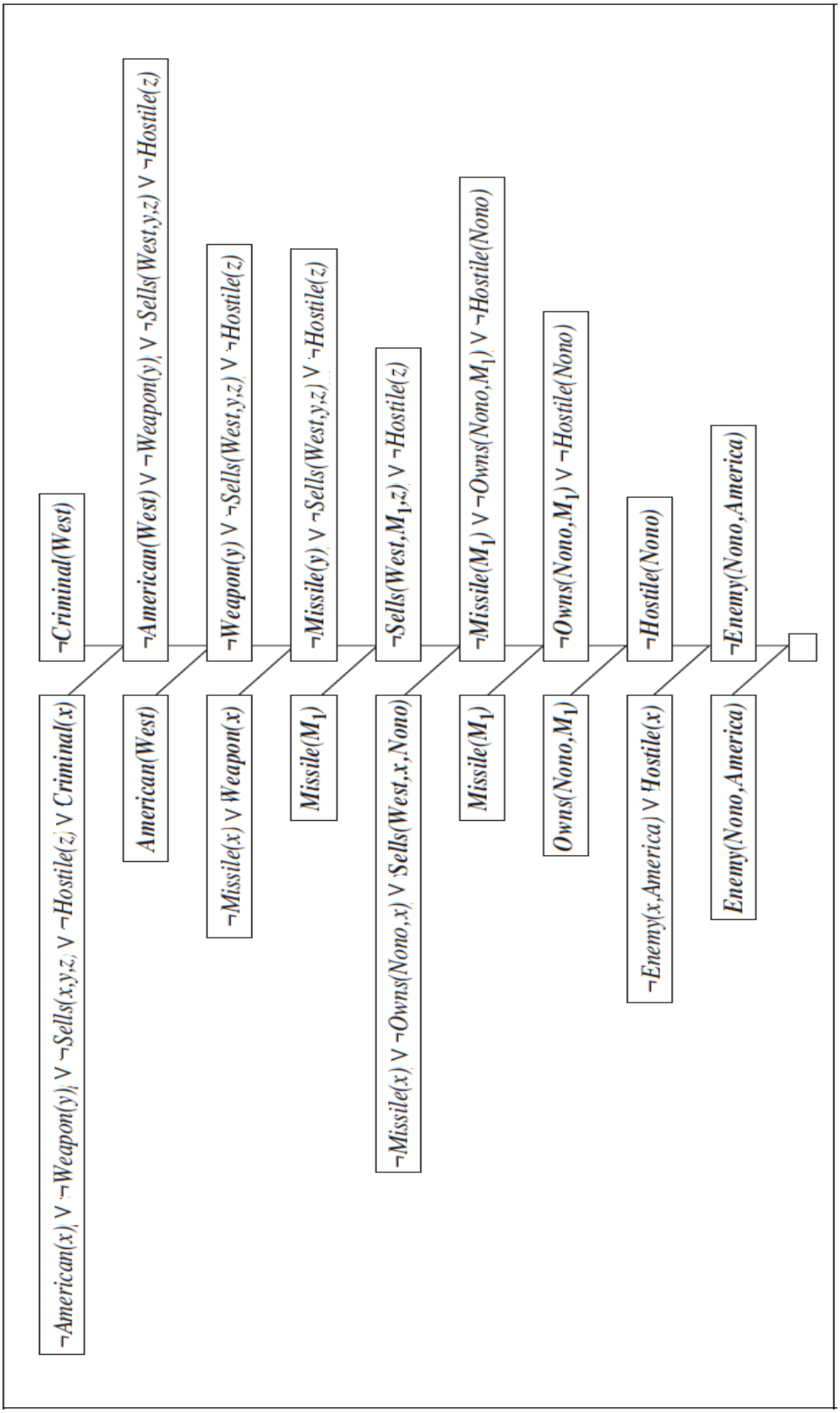
C. ∀ x Animal(x) ⇒ Loves(Jack, x)

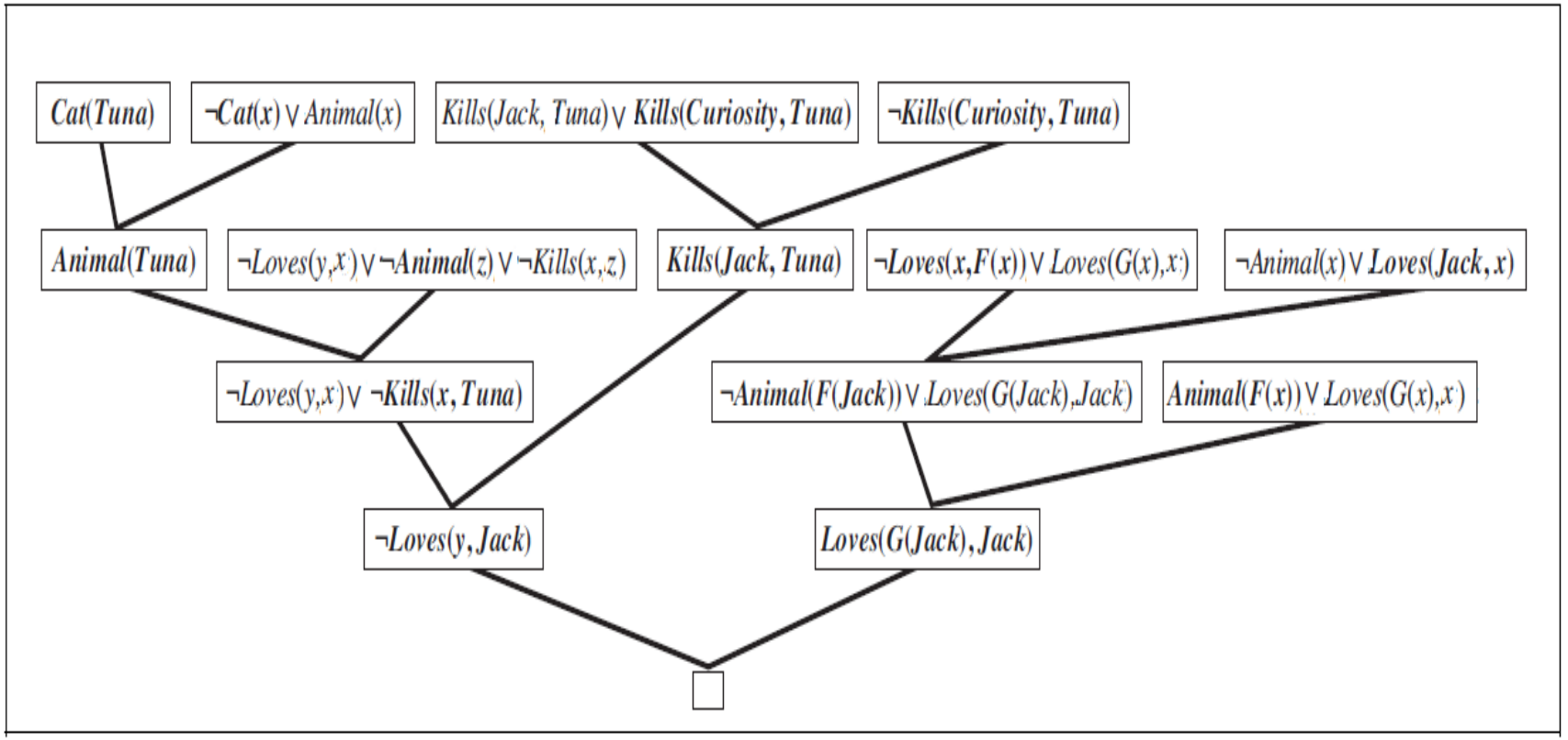
D. Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)

E. Cat(Tuna)

F. ∀ x Cat(x) ⇒ Animal (x)

¬G. ¬Kills(Curiosity, Tuna)


¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x,y,z) ∨ ¬Hostile(z) ∨ Criminal(x)
¬Criminal(West)
American(West)
¬American(West) ∨ ¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ Weapon(x)
¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ ¬Owns(Nono,x) ∨ Sells(West,x,Nono)
¬Sells(West,M1,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(M1) ∨ ¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
Owns(Nono,M1)
¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
¬Enemy(x,America) ∨ Hostile(x)
¬Hostile(Nono)
Enemy(Nono,America)
¬Enemy(Nono,America)


Cat(Tuna)
¬Cat(x) ∨ Animal(x)
Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
¬Kills(Curiosity, Tuna)
Animal(Tuna)
¬Loves(y,x) ∨ ¬Animal(z) ∨ ¬Kills(x,z)
Kills(Jack, Tuna)
¬Loves(x,F(x)) ∨ Loves(G(x),x)
¬Animal(x) ∨ Loves(Jack, x)
¬Loves(y,x) ∨ ¬Kills(x, Tuna)
¬Animal(F(Jack)) ∨ Loves(G(Jack),Jack)
Animal(F(x)) ∨ Loves(G(x),x)
¬Loves(y, Jack)
Loves(G(Jack), Jack)

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

A1. Animal(F(x)) ∨ Loves(G(x), x)

A2. ¬Loves(x, F(x)) ∨ Loves(G(x), x)

B. ¬Loves(y, x) ∨ ¬Animal (z) ∨ ¬Kills(x, z)

C. ¬Animal(x) ∨ Loves(Jack, x)

D. Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)

E. Cat(Tuna)

F. ¬Cat(x) ∨ Animal (x)

¬G. ¬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.

Completeness of resolution

Resolution is **refutation-complete**, which means that *if* a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Hence, it can be used to find all answers to a given question, Q(x), by proving that KB ∧ ¬Q(x) is unsatisfiable.

The basic structure of the proof (Figure 9.13) is as follows:

1. First, we observe that if S is unsatisfiable, then there exists a particular set of *ground instances* of the clauses of S such that this set is also unsatisfiable.

2. We then appeal to the **ground resolution theorem**, which states that propositional resolution is complete for ground sentences.

3. We then use a **lifting lemma** to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.

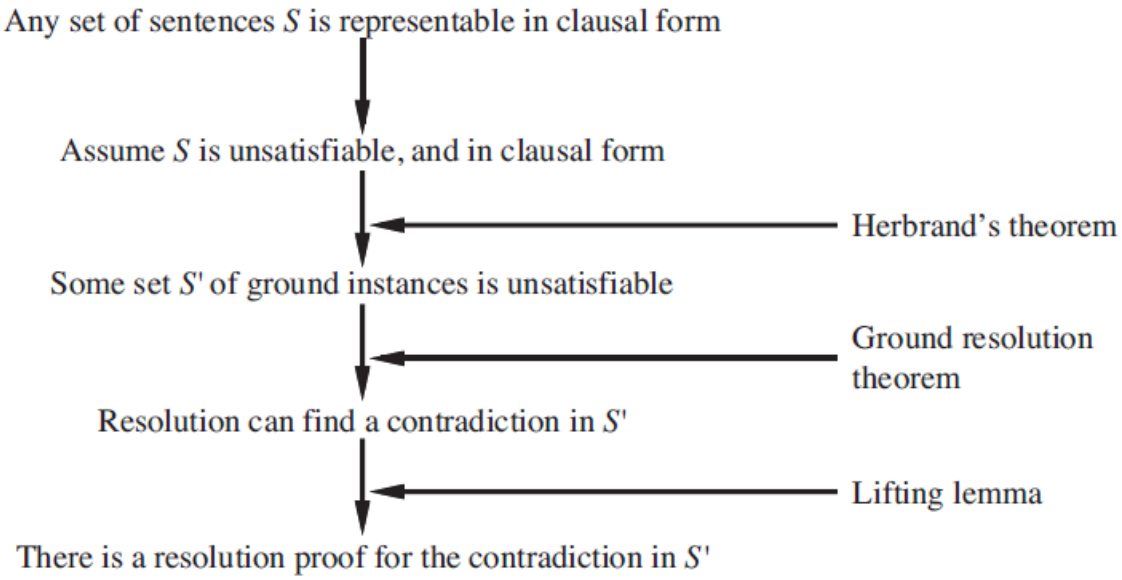


Figure 9.13 Structure of a completeness proof for resolution.

Resolution strategies

- **Unit preference**: This strategy prefers to do resolutions where one of the sentences is a single Literal, also known as a **unit clause**.
- **Set of support:** Every resolution step involve at least one element of a *set of support*. The resolvent is then added into the set of support.
- **Input resolution**: Every resolution combines one of the input sentences (from the KB or the query) with some other sentence.
- **Subsumption**: Eliminates all sentences that are subsumed by an existing sentence in the KB.

3.7 KNOWLEDGE REPRESENTATION

The steps in Knowledge representation are:

1. Identify the task.
2. Assemble the relevant knowledge.
3. Decide on a vocabulary of predicates, functions, and constants.
4. Encode general knowledge about the domain.
5. Encode a description of the specific problem instance.
6. Pose queries to the inference procedure and get answers.
7. Debug the knowledge base.

3.7.1 Characteristics of Knowledge Representation

Representational Adequacy – The ability to represent all kinds of knowledge that are needed in that domain.

Inferential Adequacy – The ability to manipulate the structure in such a way to derive new structure corresponding to new knowledge inferred from old.

Inferential Efficiency – The ability to incorporate additional information to focus the mechanism of inference mechanism in the most promising directions.

Acquisitional Efficiency – The ability to acquire new information easily.

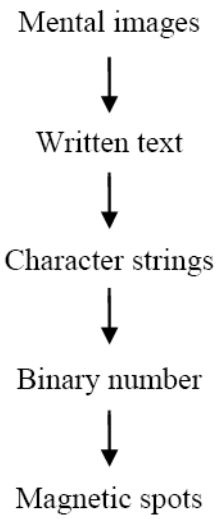


Figure: Different levels of knowledge Representation

Multiple techniques for knowledge representation are

1. Simple Relation Knowledge – declarative facts in the form of table.
2. Inheritable Knowledge – elements of specific classes inherit attributes and values from more general classes in which they are included.
3. Inferential Knowledge – implements standard logical rules of inference with resolution.
4. Procedural Knowledge – specifies what to do and when.

Components of Good Representation

For analysis purposes it is useful to be able to break any knowledge representation down into four fundamental components:

• Lexical Part – Determines symbols or words used

• Strutural or syntactic part – constraints on how the words or symbols used

• Semantic Part – association of real world meaning with representation.

• Procedural Part – Access procedures that generates and compute things with the representation.

Knowledge can be represented in different forms, as mental images in one‟s thoughts, as spoken or written words in some language, as graphical or other pictures, and as character strings or collections of magnetic spots stored in a computer.

3.8 ONTOLOGICAL ENGINEERING

Representing abstract concepts such as *Events*, *Time*, *Physical Objects*, and *Beliefs* that occur in many different domains is called **ontological engineering**. Normally, we cannot actually write a complete description of everything that would be far too much for even a 1000-page textbook. So we will leave placeholders where new knowledge for any domain can fit in. For example, we will define what it means to be a physical object, and the details of different types of objects, robots, televisions, books, or whatever can be filled in later. 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.

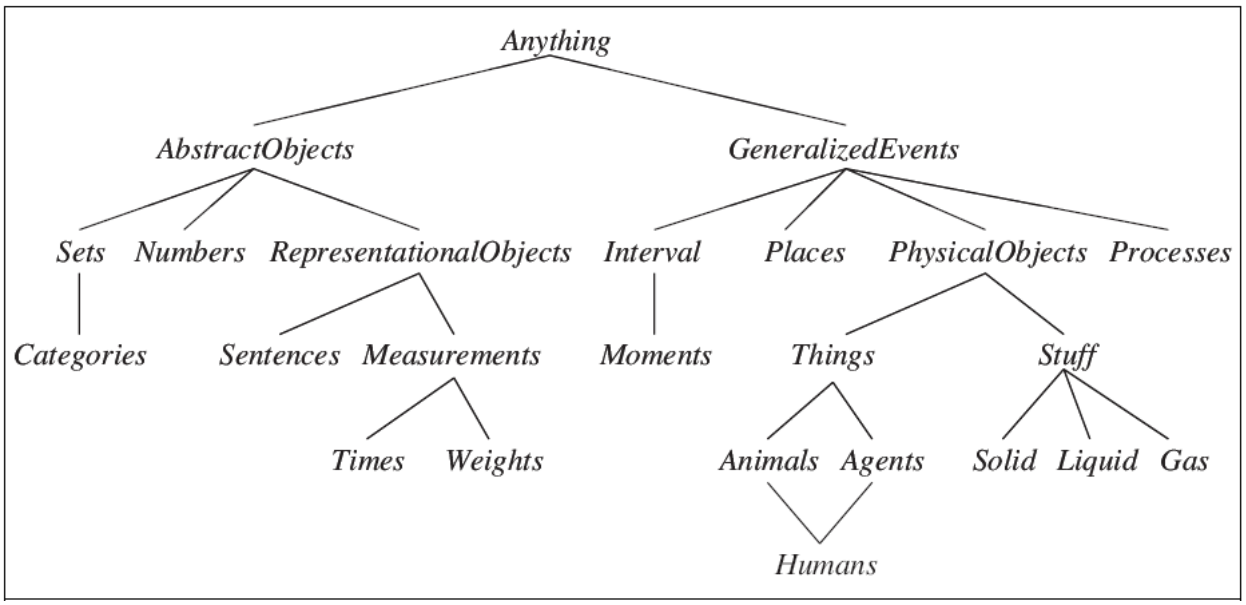


Figure 10.1 Upper Ontology

In the real world problems, nothing happens except when the agent acts, and all changes are instantaneous. A more general ontology, better suited for the real world, would allow for simultaneous changes extended over time. Each agent is having different properties, so we would need to build up agents to help predict behaviour of agent from scanty clues.

Two major characteristics of general-purpose ontologies distinguish them from collections of special-purpose ontologies:

- A general-purpose ontology should be applicable in more or less any special-purpose domain. This means that no representational issue can be finessed or brushed under the carpet.
- In any sufficiently demanding domain, different areas of knowledge must be unified, because reasoning and problem solving could involve several areas simultaneously. A robot circuit-repair system, for instance, needs to reason about circuits in terms of electrical connectivity and physical layout, and about time, both for circuit timing analysis and estimating labour costs. The sentences describing time therefore must be capable of being combined with those describing spatial layout and must work equally well for nano seconds and minutes and for angstroms and meters.

3.9 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. Categories also serve to make predictions about objects once they are classified. One infers the presence of certain objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects.

For example, from its green and yellow mottled skin, one-foot diameter, ovoid shape, red flesh, black seeds, and presence in the fruit aisle, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad. 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 **reif** 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. We will use subcategory, subclass, and subset interchangeably. Categories serve to organize and simplify the knowledge base through **inheritance**.

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

- An object is a member of a category. BB ∈ Basketballs
- A category is a subclass of another category. Basketballs ⊂ Balls
- All members of a category have some properties. x ∈ Basketballs ⇒ Round (x)
- Members of a category can be recognized by some properties. Orange(x) ∧ Round (x) ∧ Diameter(x)=9.5'' ∧ x∈ Balls ⇒ x∈ Basketballs
- A category as a whole has some properties. Dogs ∈ DomesticatedSpecies

We say that two or more categories are **disjoint** if they have no members in common. And even if we know that males and females are disjoint, we will not know that an animal that is not a male must be a female, unless we say that males and females constitute an exhaustive decomposition of the animals. A disjoint exhaustive decomposition is known as a partition. The following examples illustrate these three concepts:

Disjoint({Animals,Vegetables})

ExhaustiveDecomposition({Americans,Canadians, Mexicans}, NorthAmericans)

Partition({Males, Females}, Animals)

3.9.1 Physical composition:

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

PartOf (EasternEurope, Europe)

PartOf (Europe, Earth).

The PartOf relation is transitive and reflexive; that is,

PartOf (x, y) ∧ PartOf (y, z) ⇒ PartOf (x, z) .

PartOf (x, x)

Therefore, we can conclude PartOf (EasternEurope , Earth).

We can define BunchOf in terms of the PartOf relation. ∀x x ∈ s ⇒ 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:

∀ y [∀x x ∈ s ⇒ PartOf (x, y)] ⇒ PartOf (BunchOf (s), y) .

These axioms are an example of a general technique called **logical minimization,** which means defining an object as the smallest one satisfying certain conditions.

3.9.2 Measurements

Objects have height, mass, cost, and so on. The values that we assign for these properties are called **measures**. The same length has different names in our language. We represent the length with a units function that takes a number as argument. If the line segment is called L1, we can write Length(L1)=Inches(1.5)=Centimeters(3.81) . Conversion between units is done by equating multiples of one unit to another: Centimeters(2.54 × d)=Inches(d) .

3.9.3 Objects: Things and stuff

There is, a significant portion of reality that seems to defy any obvious **individuation,** division into distinct objects. We give this portion the generic name **stuff.** Linguists distinguish between count nouns, such as balls, holes, and theorems, and mass nouns, such as butter, water, and energy. Several competing ontologies claim to handle this distinction. Here we describe just one.

$$B \in Butter \wedge PartOf(p, b) \Rightarrow p \in Butter$$

There are some properties that are **intrinsic,** they belong to the very substance of the object, rather than to the object as a whole. When you cut an instance of stuff in half, the two pieces retain the intrinsic properties things like density, boiling point, flavor, color, ownership, and so on. On the other hand, their **extrinsic** properties, weight, length, shape, and so on are not retained under subdivision.

3.10 EVENTS

Situations denote the states resulting from executing actions. This approach is called **situation calculus. Situations** are logical terms consisting of the initial situation and all situations that are generated by applying an action to a situation. **Fluents** are functions and predicates that vary from one situation to the next, such as the location the agent.

Event calculus, is based on points of time rather than on situations. An agent should be able to deduce the outcome of a given sequence of actions is called **projection** task. With a suitable constructive inference algorithm, agent should be able to find a sequence that achieves a desired effect called **planning** tasks. The complete set of predicates for one version of the event calculus is T(f, t) where Fluent f is true at time t. Formally, the axioms are:

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

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

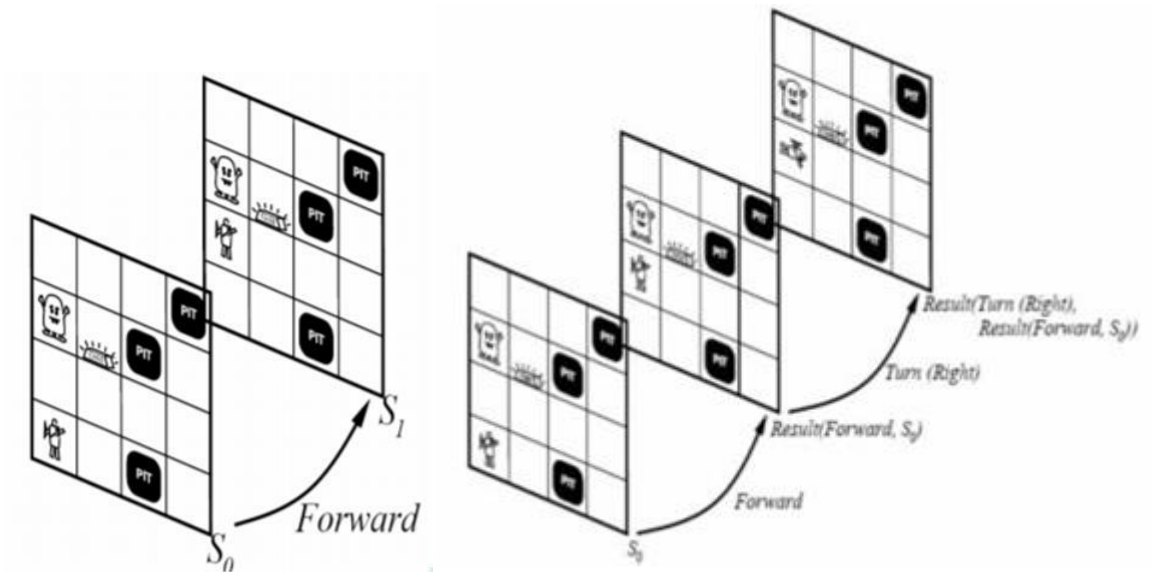


Figure 3.10.1 Situation Calculus

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.

3.10.1 Processes

Discrete events have a definite structure. For example, Shankar's trip has a beginning, middle, and end. Categories of events with continuously happening property are called **process** categories **or liquid event** categories. Any process e that happens over an interval also happens over any subinterval:

$$(e \in \text{Processes}) \wedge \text{Happens}(e, (t1, t4)) \wedge (t1 < t2 < t3 < t4) \Rightarrow \text{Happens}(e, (t2, t3))$$

For example, the predicate T, represents time. Here it is some subinterval of time lunch hour.

T(Working (Stuart), TodayLuchHour)

This liquid event is called **temporal substance**, whereas substance like butter is **spatial substance**.

3.10.2 Time intervals

Time is important to any agent that takes action, and there have been much work to represent time intervals. The function Duration gives the difference between the end time and the start time.

$$\text{Interval}(i) \Rightarrow \text{Duration}(i) = (\text{Time}(\text{End}(i)) - \text{Time}(\text{Begin}(i))) .$$

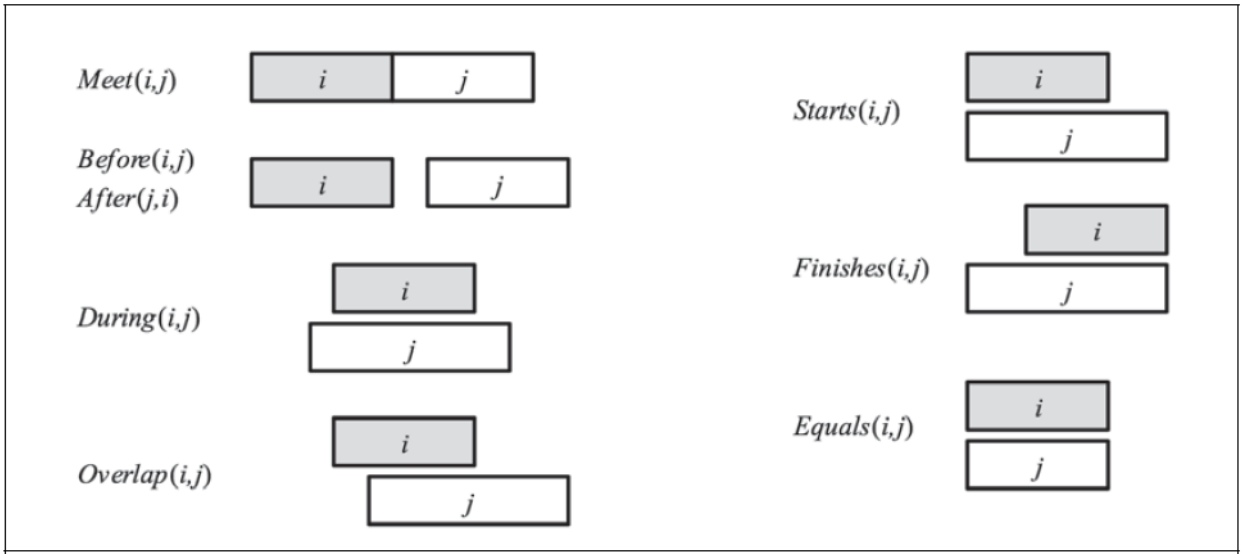


Figure 12.2 Predicates on time intervals.

The complete set of interval relations, is shown graphically in Figure 12.2 and logically below:

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

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

$$\text{After}(j, i) \Leftrightarrow \text{Before}(i, j)$$

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

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

$$\text{Begins}(i, j) \Leftrightarrow \text{Begin}(i) = \text{Begin}(j)$$

$$\text{Finishes}(i, j) \Leftrightarrow \text{End}(i) = \text{End}(j)$$

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

To say that the reign of Elizabeth II immediately followed that of George VI, and the reign of Elvis overlapped with the 1950s, we can write the following:

$$\text{Meets}(\text{ReignOf}(\text{GeorgeVI}), \text{ReignOf}(\text{ElizabethII})) .$$

$$\text{Overlap}(\text{Fifties}, \text{ReignOf}(\text{Elvis})) .$$

$$\text{Begin}(\text{Fifties})=\text{Begin}(\text{AD1950}) .$$

$$\text{End}(\text{Fifties})=\text{End}(\text{AD1959}) .$$

3.10.3 Fluents and objects

Physical objects can be viewed as generalized events, in the sense that a physical object is chunk of space–time. We can describe the changing properties of USA using state fluents, such

as Population (USA). A property of the USA that changes every four or eight years is its president. To say that George Washington was president throughout 1790, we can write

T(Equals(President(USA), GeorgeWashington),AD1790) .

We use the function symbol Equals rather than the standard logical predicate =, because we cannot have a predicate as an argument to T, and because the interpretation is *not* that GeorgeWashington and President(USA) are logically identical.

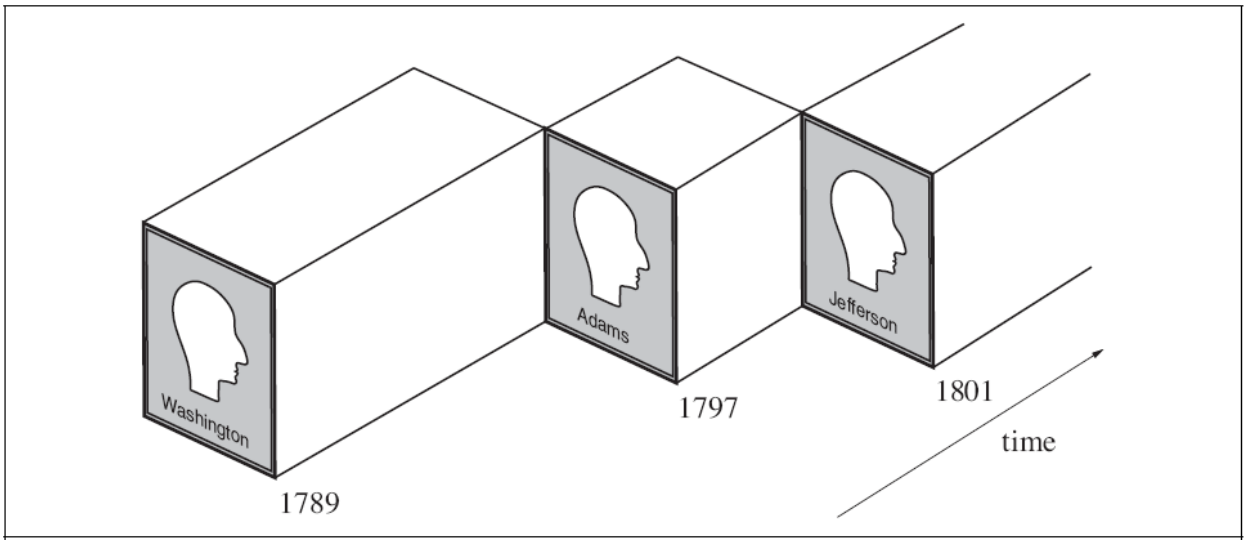


Figure 12.3 A schematic view of the object President (USA) for the first 15 years of its existence.

3.11 MENTAL EVENTS AND MENTAL OBJECTS

The agents we have constructed so far have beliefs and can deduce new beliefs. 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.

3.11.1 Knowledge and Belief

Plato defines knowledge as "knowledge is justified true belief". That is, if it is true, if you believe it, and if you have an unassailably good reason, then you know it. That means that if you know something, it must be true, and we have the axiom: **K**aP ⇒ P .

Furthermore, logical agents should be able to introspect on their own knowledge. If they know something, then they know that they know it: **K**aP ⇒ **K**a(**K**aP) . We can define similar axioms for belief (often denoted by **B**) and other modalities.

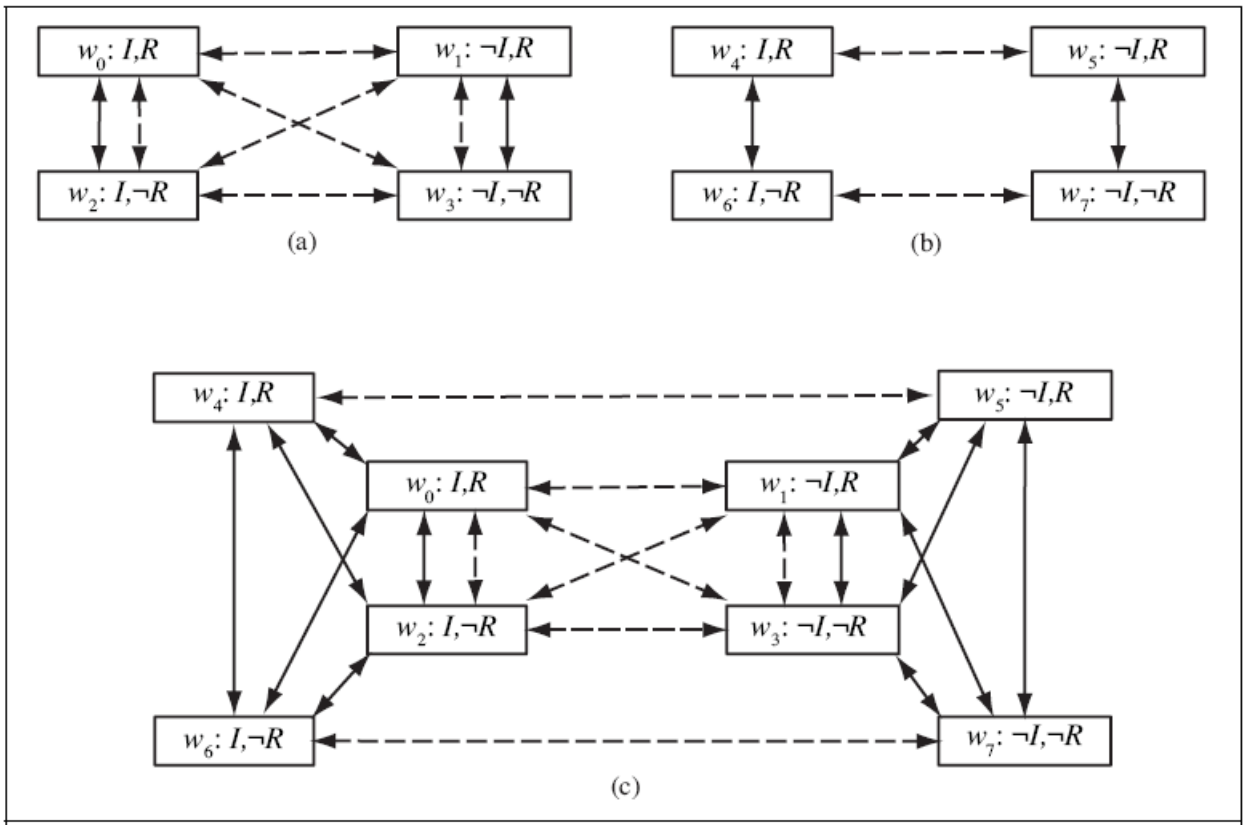


Figure 12.4 Possible worlds with accessibility relations

If our agent knows that 2 + 2 = 4 and 4 < 5, then we want our agent to know that 2 + 2 < 5. This property is called **referential transparency.** But for propositional attitudes like *believes* and *knows*, we would like to have referential opacity. **Referential opacity** means one cannot substitute an equal term for the second argument without changing the meaning. For example, suppose we try to assert that Lois knows that Superman can fly:

Knows(Lois, CanFly(Superman))

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. But Lois don't know Clark is Superman. **Modal logic** is designed to address this problem. In modality logic, propositional attitudes such as Believes and Knows become modal operators that are referentially opaque. That is Superman is Clark, is not applicable everywhere. L**ogical omniscience** is that, if an agent knows a set of axioms, then it knows all consequences of those axioms. If a logical agent believes something, then it believes that it believes it. There have been attempts to define a form of limited rational agents, which can make a limited number of deductions in a limited time.

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. **K**Superman (solid arrows) and **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, either Superman knows I (Identity), 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, R (Rain) is predicted and in w6 she knows rain is not predicted.

3.12 REASONING SYSTEMS FOR CATEGORIES

Reasoning Systems for Categories 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.

3.12.1 Semantic networks

In 1909, Charles S. Peirce proposed a convenient graphical notation of nodes and edges called **existential graphs** that he called "the logic of the future." There are many variants of semantic networks, but all are capable of representing individual objects, categories of objects, and relations among objects.

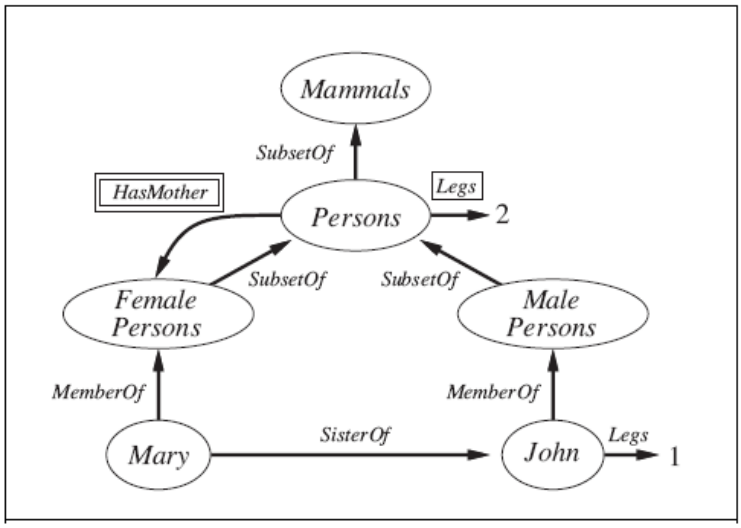


Figure 12.5 A semantic network with four objects and four categories.

A typical graphical notation displays object or category names in ovals or boxes, and connects them with labelled 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.

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 $\forall x\ x \in \text{Persons} \Rightarrow [\forall y\ \text{HasMother}(x, y) \Rightarrow y \in \text{FemalePersons}]$

The semantic network notation makes it convenient to perform **inheritance.** Persons inherits properties of mammals. 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**. In such cases, the inheritance algorithm might find two or more conflicting values answering the query. For this reason, multiple inheritance is banned in some **object-oriented programming** (OOP) languages, such as Java. It is usually allowed in semantic networks. **Inverse Links** is a form of inference, for example HasSister is the inverse of SisterOf.

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.

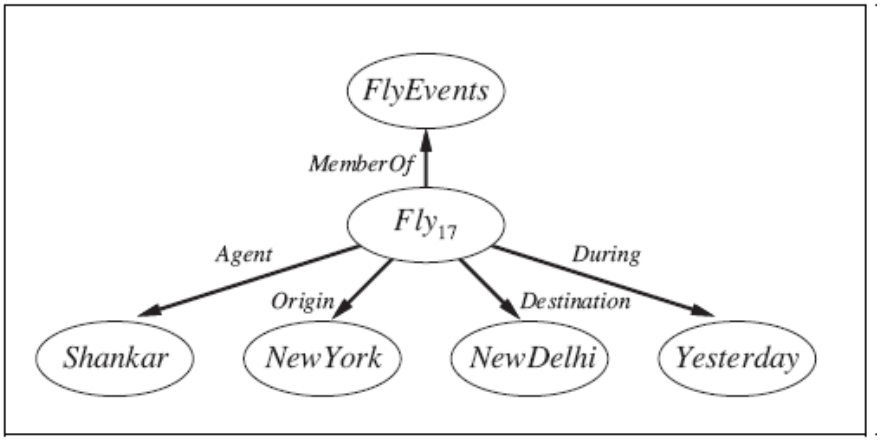


Figure 12.6 A semantic network of Fly(Shankar , NewYork, NewDelhi ,Yesterday).

3.12.1.1 Advantages of Semantic Networks

1. Simplicity and transparency of the inference processes.
2. Designers can build large network and still have a good idea about what queries will be efficient, because
   (a) it is easy to visualize the steps that the inference procedure will go through and
   (b) in some cases the query language is so simple that difficult queries cannot be posed.
3. In cases where the expressive power proves to be too limiting, many semantic network systems provide for **procedural attachment** to fill in the gaps. Procedural attachment is a technique whereby a query about a certain relation results in a call to a special procedure designed for that relation rather than a general inference algorithm.
4. One of the most important aspects of semantic networks is their ability to represent **default values** for categories. The default is **overridden** by the more specific value.

3.12.2 Description logics

Description logics are notations that are designed to make it easier to describe definitions and properties of categories. Description logic systems evolved from semantic networks in response to retain the emphasis on taxonomic structure. The principal inference tasks for description logics are

1. **Subsumption** - checking if one category is a subset of another by comparing their definitions
2. **Classification** – checking whether an object belongs to a category.
3. **Consistency** - whether the membership criteria are logically satisfiable.

The CLASSIC language is a typical description logic. For example, to say that bachelors are unmarried adult males we would write, Bachelor = And(Unmarried, Adult ,Male) . The equivalent in first-order logic would be Bachelor (x) ⇔ Unmarried(x) ∧ Adult(x) ∧ Male(x) The description logic effectively allows direct logical operations on predicates, rather than sentences joined by connectives.

$$
\begin{array}{rcl}
Concept & \rightarrow & \textbf{Thing} \mid ConceptName \\
& \mid & \textbf{And}(Concept, \ldots) \\
& \mid & \textbf{All}(RoleName, Concept) \\
& \mid & \textbf{AtLeast}(Integer, RoleName) \\
& \mid & \textbf{AtMost}(Integer, RoleName) \\
& \mid & \textbf{Fills}(RoleName, IndividualName, \ldots) \\
& \mid & \textbf{SameAs}(Path, Path) \\
& \mid & \textbf{OneOf}(IndividualName, \ldots) \\
Path & \rightarrow & [RoleName, \ldots]
\end{array}
$$

Figure 10.11 Syntax of CLASSIC Language

3.13 REASONING WITH DEFAULT INFORMATION

A simple example of an assertion with default status is people have two legs. This default can be overridden by more specific information, such as that Long John Silver has one leg. We saw that the inheritance mechanism in semantic networks implements the overriding of defaults in a simple and natural way.

3.13.1 Open and Closed Worlds

If a university advertises “Courses offered are CS01, CS02, CS03 and EE01”, the answer for the question “How many courses are offered by the university?” is “four. But in first order logic the answer is “at least four and at most infinity”. The reason is that, the assertion do not deny the possibility that the unmentioned courses are also offered. This example shows that database and human communication are different from first order logic.

1. Database assumes information provided is complete. So the atomic statements not asserted to be true are assumed to be false. This is **Closed World Assumption (CWA)**
2. Database assume distinct names refer to distinct objects. This is called **Unique Names Assumption (UNA).**

Horn clause use the idea of **negation as failure.** The idea is that negative literal, not P, can be proved true just in case the proof of P fails. Consider the assertion P is not Q. This has two minimal models, P and Q. An alternative idea is **stable model,** which is a minimal model where every atom in the model has a justification, with H **reduct** with respect to M. The reduct of P ← not Q with respect to P is a minimal model { P }. Therefore P is a stable model.

3.13.2 Circumscription and default logic

We have seen natural reasoning processes violate the **monotonicity property** of logic. Monotonicity requires all entailed sentences to remain entailed after new sentences are added to the KB. That is, if KB |= α then KB ∧ β |= α. Under the closed-world assumption, if a proposition α is not mentioned in KB then KB |= ¬α, but KB ∧ α |= α.

These failures of monotonicity are widespread in common sense reasoning. Because humans often "jump to conclusions." For example, when one sees a car parked on the street, one is normally willing to believe that it has four wheels even though only three are visible. If new evidence arrives - for example, if one sees the owner carrying a wheel and notices that the car is jacked up - then the conclusion can be retracted. This kind of reasoning is said to exhibit **non-monotonicity**, because the set of beliefs does not grow monotonically over time as new evidence arrives. **Nonmonotonic logics** have been devised with modified notions of truth and entailment in order to capture such behaviour. We will look at two such logics that have been studied extensively: **circumscription and default logic.**

3.13.3 Circumscription

The idea is to specify particular predicates that are assumed to be "as false as possible"—that is, false for every object except those for which they are known to be true. For example, suppose we want to assert the default rule that birds fly. We would introduce a predicate, say Abnormal $_1$(x), and write

$$\text{Bird}(x) \wedge \neg \text{Abnormal}_1(x) \Rightarrow \text{Flies}(x) .$$

If we say that Abnormal $_1$ is to be circumscribed, a circumscriptive reasoner is entitled to assume ¬Abnormal $_1$(x) unless Abnormal $_1$(x) is known to be true. This allows the conclusion Flies(Tweety) to be drawn from the premise Bird(Tweety), but the conclusion no longer holds if Abnormal $_1$(Tweety) is asserted. For circumscription, one model is preferred to another if it has fewer abnormal objects.

Let us see how this idea works in the context of multiple inheritance in semantic networks. The standard example for which multiple inheritance is problematic is called the "Nixon diamond." It arises from the observation that Richard Nixon was both a Quaker (and hence by default a pacifist) and a Republican (and hence by default not a pacifist). We can write this as follows:

Republican(Nixon) ∧ Quaker(Nixon) .

Republican(x) ∧¬Abnormal 2(x) ⇒ ¬Pacifist (x) .

Quaker(x) ∧ ¬Abnormal 3(x) ⇒ Pacifist (x) .

If we circumscribe Abnormal 2 and Abnormal 3, there are two preferred models: one in jwhich Abnormal 2(Nixon) and Pacifist (Nixon) hold and one in which Abnormal 3(Nixon) and ¬Pacifist(Nixon) hold. Thus, the circumscriptive reasoner remains properly agnostic as to whether Nixon was a pacifist. If we wish, in addition, to assert that religious beliefs take PRIORITIZED precedence over political beliefs, we can use a formalism called **prioritized circumscription** to give preference to models where Abnormal 3 is minimized.

3.13.4 Default Logic

Default logic is a formalism in which **default rules** can be written to generate contingent, non-monotonic conclusions. A default rule looks like this: Bird(x) : Flies(x)/Flies(x) . This rule means that if Bird(x) is true, and if Flies(x) is consistent with the knowledge base, then Flies(x) may be concluded by default. The Nixon-diamond example can be represented in default logic with one fact and two default rules:

Republican(Nixon) ∧ Quaker(Nixon) .

Republican(x) : ¬Pacifist (x)/¬Pacifist (x) .

Quaker(x) : Pacifist (x)/Pacifist (x) .

3.13.5 Truth Maintenance Systems (TMS)

Many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain. Inevitably, some of these inferred facts will turn out to be wrong and will have to be retracted in the face of new information. This process is called **belief revision.**

Suppose that a knowledge base KB contains a sentence P - perhaps a default conclusion recorded by a forward-chaining algorithm, or perhaps just an incorrect assertion - and we want to execute TELL(KB, ¬P). To avoid creating a contradiction, we must first execute RETRACT(KB, P). **Truth maintenance systems**, or TMSs, are designed to handle exactly these kinds of complications.

One simple approach to truth maintenance is to keep track of the order in which sentences are told to the knowledge base by numbering them from P1 to Pn. When the call RETRACT(KB, Pi) is made, the system reverts to the state just before Pi was added, thereby removing both Pi and any inferences that were derived from Pi. For systems to which many facts are being added, such as large commercial databases this is impractical.

1. **Justification-Based Truth Maintenance System**, or **JTMS**. In a JTMS, each sentence in the knowledge base is annotated with a **justification** consisting of the set of sentences from which it was inferred. RETRACT(P) will delete exactly those sentences for which P is a member of every justification. So, if a sentence Q had the single justification {P, $P \Rightarrow Q$}, it would be removed; if it had the additional justification {P, $P \vee R \Rightarrow Q$}, it would still be removed; but if it also had the justification {R, $P \vee R \Rightarrow Q$}, then it would be spared.
2. An **Assumption-Based Truth Maintenance System,** or **ATMS**, represents all the states that have ever been considered at the same time. Whereas a JTMS simply labels each sentence as being in or out, an ATMS keeps track, for each sentence, of which assumptions would cause the sentence to be true. In other words, each sentence has a label that consists of a set of assumption sets.

PART A (2 MARK QUESTIONS)

1. Define a knowledge Base:

Knowledge base is the central component of knowledge base agent and it is described as a set of representations of facts about the world.

2. Define a Sentence?

Each individual representation of facts is called a sentence. The sentences are expressed in a language called as knowledge representation language.

3. Define an inference procedure

An inference procedure reports whether or not a sentence is entiled by knowledge base provided a knowledge base and a sentence .An inference procedure 'i' can be described by the sentences that it can derive. If i can derive from knowledge base, we can write. KB --Alpha is derived from KB or i derives alpha from KB.

4. What are the three levels in describing knowledge based agent?

1) Logical level 2) Implementation level 3) Knowledge level or epistemological level

5. Define Syntax?

Syntax is the arrangement of words. Syntax of knowledge describes the possible configurations that can constitute sentences. Syntax of the language describes how to make sentences.

6. Define Semantics

The semantics of the language defines the truth of each sentence with respect to each possible world. With this semantics, when a particular configuration exists within an agent, the agent believes the corresponding sentence.

7. Define Logic

Logic is one which consist of

i. A formal system for describing states of affairs, consisting of a) Syntax b)Semantics.

ii. Proof Theory – a set of rules for deducing the entailment of a set sentences.

8. What is entailment

The relation between sentences is called entailment. The formal definition of entailment is this: if and only if in every model in which x -> y is true, x is also true or if y is true then x must also be true. Informally the truth x is contained in the truth y.

9. What is truth Preserving

An inference algorithm that derives only entailed sentences is called sound or truth preserving .

10. Define a Proof

A sequence of application of inference rules is called a proof. Finding proof is exactly finding solution to search problems. If the successor function is defined to generate all possible applications of inference rules then the search algorithms can be applied to find proofs.

11. Define a Complete inference procedure

An inference procedure is complete if it can derive all true conditions from a set of premises.

12. Define Interpretation

Interpretation specifies exactly which objects, relations and functions are referred by the constant predicate, and function symbols.

13. Define Validity of a sentence

A sentence is valid or necessarily true if and only if it is true under all possible interpretation in all possible world.

14. Define Satistiability of a sentence

A sentence is satisfiable if and only if there is some interpretation in some world for which it is true.

15. Define true sentence

A sentence is true under a particular interpretation if the state of affairs it represents is the case.

16. What are the basic Components of propositonal logic?

i. Logical Constants (True, False)

17. Define Modus Ponen's rule in Propositional logic?

The standard patterns of inference that can be applied to derive chains of conclusions that lead to the desired goal is said to be Modus Ponen's rule.

18. Define probability distribution:

Eg. P(weather) = (0.7,0.2,0.08,0.02). This type of notations simplifies many equations.

19. What is an atomic event?

An atomic event is an assignment of particular values to all variables, in other words, the complete specifications of the state of domain.

20. What Is Called a Decision Theory ?

Preferences As Expressed by Utilities Are Combined with Probabilities in the General Theory of Rational Decisions Called Decision Theory.

Decision Theory = Probability Theory + Utility Theory.

PART B (13 MARK QUESTIONS)

1. Give the Syntax and Semantics of a first order logic in detail with an example
2. Give Syntax and Semantics of a first order logic for a family domain.
3. Give the Syntax and Semantics of a first order logic for Numbers, Sets, Lists domain.
4. Elaborate upon the process of knowledge engineering with electronic circuit's domain.
5. Explain about unification with an algorithm in a first order logic.
6. Explain in detail the concept of theorem proverbs.
7. Explain forward chaining and backward chaining in detail for a first order definite clauses. (UNIV QUES)
8. Explain how categories and objects are presented in any four sets.
9. Elaborate upon the ontology for situation calculus.
10. Elaborate upon the ontology for event calculus.
11. Explain predicate logic (UNIV QUES)
12. Write notes on proposition logic (UNIV QUES)
13. Explain the resolution procedure with an example (UNIV QUES)
14. Illustrate use of predicate logic to represent knowledge with suitable example (UNIV QUES)
15. How facts are represented using prepositional logic? Give an example (UNIV QUES)