UNIT - III

Logic and Knowledge Representation

First-Order Logic: Representation, Syntax and Semantics of First-Order Logic, Using First-Order Logic, Knowledge Engineering in First-Order Logic.

Inference in First-Order Logic: Propositional vs. First-Order Inference, Unification and Lifting, 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.

LOGIC AND KNOWLEDGE REPRESENTATION:

1. INTRODUCTION:

- In the topic of Propositional logic, we have seen that how to represent statements using propositional logic.
- But unfortunately, in propositional logic, we can only represent the facts, which are either true or false.
- PL is not sufficient to represent the complex sentences or natural language statements.
- The propositional logic has very limited expressive power.
- Consider the following sentence, which we cannot represent using PL logic.

  "Some humans are intelligent", or

  "Sachin likes cricket."

- To represent the above statements, PL logic is not sufficient, so we required some more powerful logic, such as first-order logic.

FIRST-ORDER LOGIC:

- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
- FOL is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as **Predicate logic or First-order predicate logic**. First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.
- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:

  i. **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, Wumpus,

  ii. **Relations: It can be unary relation such as:** red, round, is adjacent, **or n-any relation such as:** the sister of, brother of, has color, comes between

  iii. **Function:** Father of, best friend, third inning of, end of,

  - As a natural language, first-order logic also has two main parts:

    a. **Syntax**

    b. **Semantics**

2. SYNTAX OF FIRST-ORDER LOGIC:

- The syntax of FOL determines which collection of symbols is a logical expression in first-order logic. The basic syntactic elements of first-order logic are symbols. We write statements in short-hand notation in FOL.

Example: John likes all kind of food

Conversion of Facts into FOL.

Basic Elements of First-order logic:

Following are the $\forall x: food(x) \rightarrow likes(John, x)$ basic elements of FOL syntax:

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf, ...
Connectives	$\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$
Equality	==
Quantifier	$\forall$, $\exists$

i. Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2... term n).**

Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay).

Chinky is a cat: => cat (Chinky).

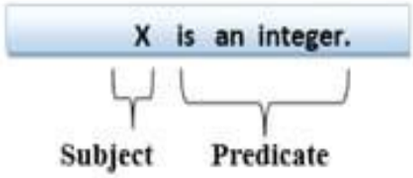
ii. Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Consider the statement: "x is an integer." it consists of two parts, the first part x is the subject of the statement and second part "is an integer" is known as a predicate.



Quantifiers in First-order logic:

- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:

  a. **Universal Quantifier, (for all, everyone, everything)**

  b. **Existential quantifier, (for some, at least one).**

a. **Universal Quantifier:**

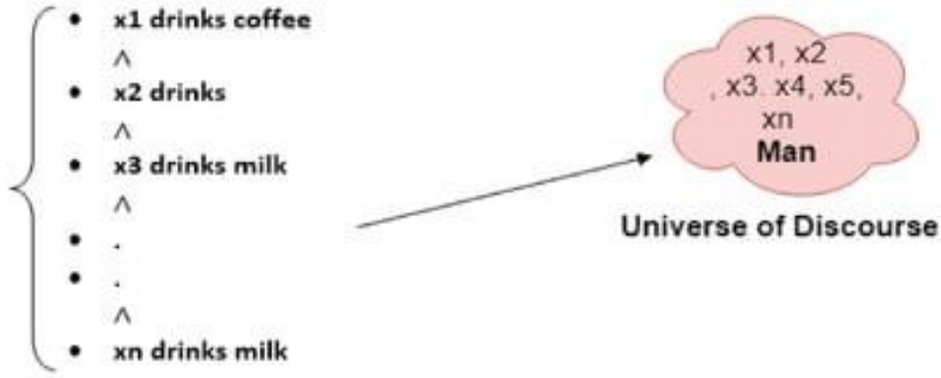
- Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.
- The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.
- If x is a variable, then ∀x is read as:
  - **For all x**
  - **For each x**
  - **For every x.**

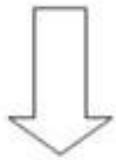
Example:

- **All man drink coffee.**

  Solution:

  Let a variable x which refers to a cat so all x can be represented in UOD as below:





So in shorthand notation, we can write it as :

∀x man(x) → drink (x, coffee).

It will be read as: There are all x where x is a man who drink coffee.

b. Existential Quantifier:

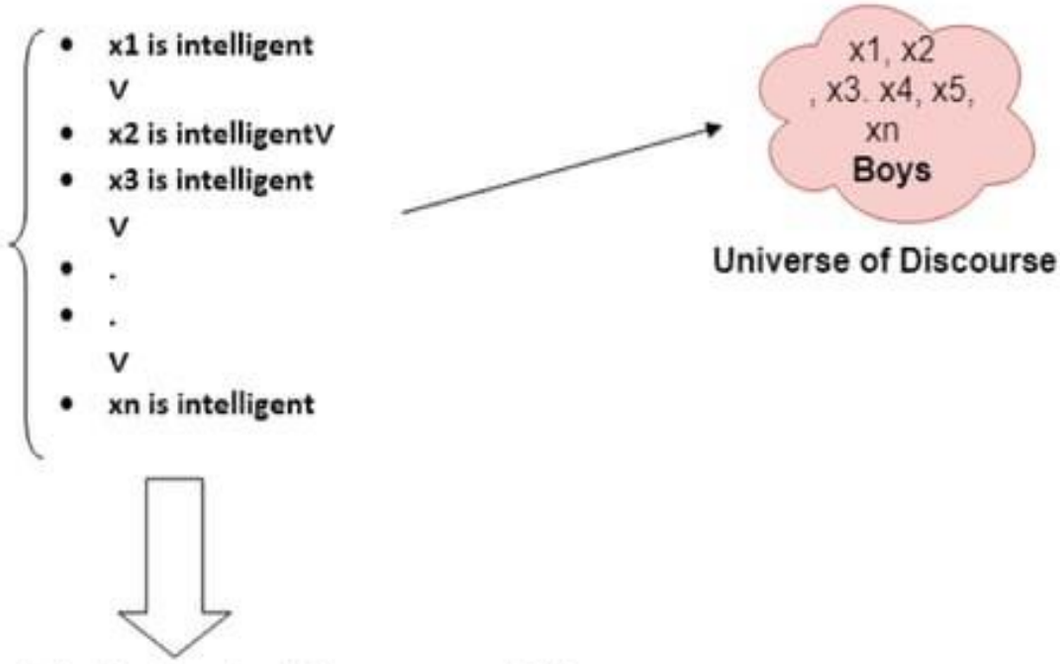
- Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.
- It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.
- If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

  There exists a 'x.'

  For some 'x.'

  For at least one 'x.'

Some boys are intelligent.



So in short-hand notation, we can write it as:

$\exists x: boys(x) \wedge intelligent(x)$

It will be read as: There are some x where x is a boy who is intelligent.

3. USING FIRST ORDER LOGIC:

Points to remember:

- The main connective for universal quantifier ∀ is implication →.
- The main connective for existential quantifier ∃ is and ∧.

Properties of Quantifiers:

- In universal quantifier, ∀x∀y is similar to ∀y∀x.
- In Existential quantifier, ∃x∃y is similar to ∃y∃x.
- ∃x∀y is not similar to ∀y∃x.

Some Examples of FOL using quantifier:

1. All birds fly.

In this question the predicate is "**fly(bird)**."

And since there are all birds who fly so it will be represented as follows.

$\forall x$ bird(x) $\rightarrow$fly(x).

2. Every man respects his parent.

In this question, the predicate is **"respect(x, y)," where x=man, and y= parent**.

Since there is every man so will use $\forall$, and it will be represented as follows:

$\forall x$ man(x) $\rightarrow$ respects (x, parent).

3. Some boys play cricket.

In this question, the predicate is "**play(x, y),**" where x= boys, and y= game. Since there are some boys so we will use **$\exists$, and it will be represented as**:

$\exists x$ boys(x) $\rightarrow$ play(x, cricket).

4. Not all students like both Mathematics and Science.

In this question, the predicate is **"like(x, y)," where x= student, and y= subject**.

Since there are not all students, so we will use **$\forall$ with negation, so** following representation for this:

$\neg\forall$ (x) [student(x) $\rightarrow$ like(x, Mathematics) $\wedge$ like(x, Science)].

5. Only one student failed in Mathematics.

In this question, the predicate is **"failed(x, y)," where x= student, and y= subject**.

Since there is only one student who failed in Mathematics, so we will use following representation for this:

$\exists$(x) [student(x) $\rightarrow$ failed (x, Mathematics) $\wedge\forall$ (y) [$\neg$(x==y) $\wedge$ student(y) $\rightarrow$ $\neg$failed (x, Mathematics)].

Free and Bound Variables:

The quantifiers interact with variables which appear in a suitable way. There are two types of variables in First-order logic which are given below:

Free Variable: A variable is said to be a free variable in a formula if it occurs outside the scope of the quantifier.

Example: $\forall x \exists(y)$[P (x, y, z)], where z is a free variable.

Bound Variable: A variable is said to be a bound variable in a formula if it occurs within the scope of the quantifier.

Example: $\forall x$ [A (x) B(y)], here x and y are the bound variables.

Example of FOL:

1. Bhaskar likes aeroplanes.	Likes (bhaskar, aeroplanes).
2. Ravi's father is rani's father.	Father(father(ravi), rani)).
3. Plato is a man.	Man (plato).
4. Ram likes mango.	Likes(ram, mango).
5. Sima is a girl.	Girl(sima) .
6. Rose is red.	Red (rose).
7. John owns gold.	Owns(john, gold).
8. Ram is taller than mohan.	Taller(ram, mohan).
9. My name is khan.	Name (khan) or Name(my, khan)
10. Apple is fruit.	Fruit(apple).

11. Ram is male.	Male (ram).
12. Tuna is fish.	Fish (tuna).
13. Dashrath is ram's father.	Father (dashrath, ram).
14. Kush is son of ram.	Son(kush, ram).
15. Kaushaliya is wife of Dashrath.	Wife(kaushaliya, dashrath).
16. Clinton is tall.	Tall(clinton).
17. There is a white alligator.	Alligator (white).
18. All kings are person.	$\forall x$: Kings(x) $\rightarrow$ Person(x).
19. Nobody loves john.	$\forall x$: $\neg$Loves(x, john).
20. Every one has a father.	$\forall x$: $\exists y$: **Father(y,x)**

Translate to predicate logic:

1. Marcus was a man.
   - Man(Marcus).
2. Marcus was a Pompeian.
   - Pompeian(marcus)
3. All Pompeian were Romans.
   - $\forall x$: Pompeian(x) $\rightarrow$ Roman(x)
4. Caesar was a ruler.
   - Ruler(Caesar)
5. All Romans were either loyal to Caesar or hated him.
   - $\forall x$: Roman(x) $\rightarrow$ LoyalTo(x,Caesar) $\vee$ Hate(x,Caesar)

6. Everyone is loyal to someone.

   $\forall x$: $\exists y$: LoyalTo(x,y)

7. People only try to assassinate rulers they aren't loyal to.

   $\forall x$: $\forall y$: Person(x) $\wedge$ Ruler(y) $\wedge$ TryAssassinate(x,y) $\rightarrow \neg$LoyalTo(x,y)]

8. Marcus tried to assassinate Caesar.

   TryAssassinate(Marcus, Caesar)

9. All men are people.

   $\forall x$: Men(x) $\rightarrow$ People(x)

Example with solution:

Translate into predicate logic

i. Hari likes all kind of food.
ii. Bananas are food.
iii. Apples are food.
iv. Anything anyone eats and isn't killed by food.
v. Hari eats everything ram eats.

Solution

i. $\forall x$: Food(x) $\rightarrow$ Likes(hari, x).
ii. Food(bananas) .
iii. Food(apples) .
iv. $\forall x$: $\exists y$: Eats(y,x) $\wedge$ $\neg$Killedby(y,x) $\rightarrow$Food(x)
v. $\forall x$: eats(ram, x) $\rightarrow$ eats(hari, x) .

Example with solution:

1. Every gardener likes the sun.
2. Not Every gardener likes the sun.
3. You can fool some of the people all of the time.
4. You can fool all of the people some of the time.
5. You can fool all of the people at same time.
6. You can not fool all of the people all of the time.
7. Everyone is younger than his father.

Solution

1. $(\forall x)$: gardener(x) => likes(x, Sun)
2. ~$((\forall x)$:gardener(x) => likes(x, Sun))
3. $(\exists x):(\forall t)$:person(x) ^ time(t) => can-be-fooled(x, t)
4. $(\forall x):(\exists t)$:person(x) ^ time(t) => can-be-fooled(x ,t)
5. $(\exists t):(\forall x)$:person(x) ^ time(t) => can-be-fooled(x, t)
6. ~$((\forall x):(\forall t)$: person(x) ^ time(t) => can-be-fooled(x, t))
7. $(\forall x)$:person(x) => younger(x, father(x))

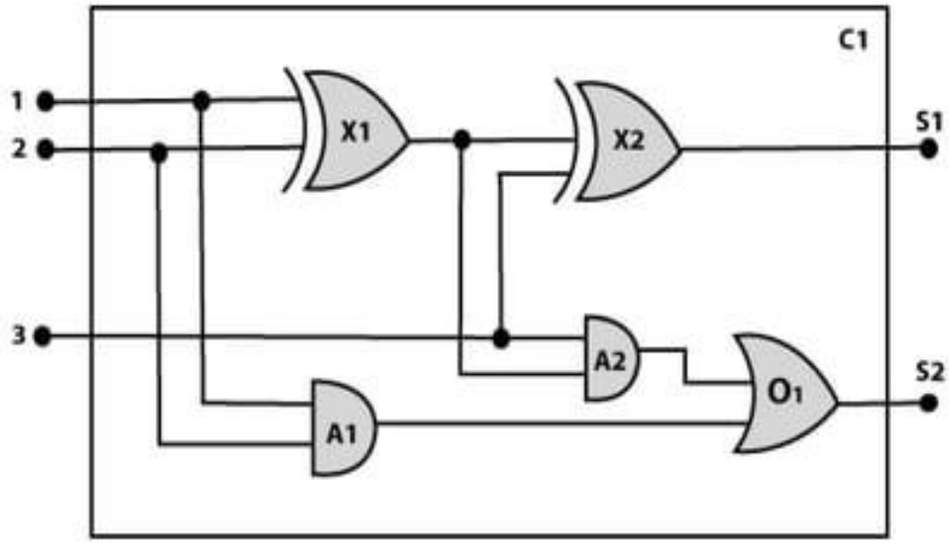
4. KNOWLEDGE ENGINEERING IN FIRST ORDER LOGIC:

What is knowledge-engineering?

- The process of constructing a knowledge-base in first-order logic is called as knowledge- engineering. In **knowledge-engineering**, someone who investigates a particular domain, learns important concept of that domain, and generates a formal representation of the objects, is known as **knowledge engineer**.
- In this topic, we will understand the Knowledge engineering process in an electronic circuit domain, which is already familiar. This approach is mainly suitable for creating **special-purpose knowledge base**.

The knowledge-engineering process:

- Following are some main steps of the knowledge-engineering process. Using these steps, we will develop a knowledge base which will allow us to reason about digital circuit (**One-bit full adder**) which is given below



1. Identify the task:

The first step of the process is to identify the task, and for the digital circuit, there are various reasoning tasks.

At the first level or highest level, we will examine the functionality of the circuit:

- **Does the circuit add properly?**
- **What will be the output of gate A2, if all the inputs are high?**

At the second level, we will examine the circuit structure details such as:

- **Which gate is connected to the first input terminal?**
- **Does the circuit have feedback loops?**

2. Assemble the relevant knowledge:

In the second step, we will assemble the relevant knowledge which is required for digital circuits. So for digital circuits, we have the following required knowledge:

- Logic circuits are made up of wires and gates.
- Signal flows through wires to the input terminal of the gate, and each gate produces the corresponding output which flows further.
- In this logic circuit, there are four types of gates used: **AND, OR, XOR, and NOT**.
- All these gates have one output terminal and two input terminals (except NOT gate, it has one input terminal).

3. Decide on vocabulary:

- The next step of the process is to select functions, predicate, and constants to represent the circuits, terminals, signals, and gates.
- Firstly we will distinguish the gates from each other and from other objects. Each gate is represented as an object which is named by a constant, such as, **Gate(X1)**.
- The functionality of each gate is determined by its type, which is taken as constants such as **AND, OR, XOR, or NOT**. Circuits will be identified by a predicate: **Circuit (C1)**.
- For the terminal, we will use predicate: **Terminal(x)**.
- For gate input, we will use the function **In (1, X1)** for denoting the first input terminal of the gate, and for output terminal we will use **Out (1, X1)**.
- The function **Arity (c, i, j)** is used to denote that circuit c has i input, j output.
- The connectivity between gates can be represented by predicate **Connect (Out (1, X1), In (1, X1))**.

We use a unary predicate **On (t)**, which is true if the signal at a terminal is on.

4. Encode general knowledge about the domain:

To encode the general knowledge about the logic circuit, we need some following rules:

- If two terminals are connected then they have the same input signal, it can be represented as:

$$\forall\ t1, t2\ \text{Terminal}(t1) \wedge \text{Terminal}(t2) \wedge \text{Connect}(t1, t2) \rightarrow \text{Signal}(t1) = \text{Signal}(2).$$

- Signal at every terminal will have either value 0 or 1, it will be represented as:

$$\forall\ t\ \text{Terminal}(t) \rightarrow \text{Signal}(t) = 1 \vee \text{Signal}(t) = 0.$$

- Connect predicates are commutative:

$$\forall\ t1, t2\ \text{Connect}(t1, t2) \rightarrow \text{Connect}(t2, t1).$$

- Representation of types of gates:

$$\forall\ g\ \text{Gate}(g) \wedge r = \text{Type}(g) \rightarrow r = \text{OR} \vee r = \text{AND} \vee r = \text{XOR} \vee r = \text{NOT}.$$

- Output of AND gate will be zero if and only if any of its input is zero.

$$\forall\ g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{AND} \rightarrow \text{Signal}(\text{Out}(1, g)) = 0 \Leftrightarrow \exists n\ \text{Signal}(\text{In}(n, g)) = 0.$$

- Output of OR gate is 1 if and only if any of its input is 1:

$$\forall\ g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{OR} \rightarrow \text{Signal}(\text{Out}(1, g)) = 1 \Leftrightarrow \exists n\ \text{Signal}(\text{In}(n, g)) = 1$$

- Output of XOR gate is 1 if and only if its inputs are different:

$$\forall\ g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{XOR} \rightarrow \text{Signal}(\text{In}(1, g)) = 1 \Leftrightarrow \text{Signal}(\text{In}(1, g)) \neq \text{Signal}(\text{In}(2, g)).$$

- Output of NOT gate is invert of its input:

$$\forall\ g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{NOT} \rightarrow \text{Signal}(\text{In}(1, g)) \neq \text{Signal}(\text{Out}(1, g)).$$

- All the gates in the above circuit have two inputs and one output (except NOT gate).

$$\forall\ g\ \text{Gate}(g) \wedge \text{Type}(g) = \text{NOT} \rightarrow \text{Arity}(g, 1, 1)$$

$$\forall\ g\ \text{Gate}(g) \wedge r = \text{Type}(g) \wedge (r = \text{AND} \vee r = \text{OR} \vee r = \text{XOR}) \rightarrow \text{Arity}(g, 2, 1).$$

- All gates are logic circuits:

$$\forall\ g\ \text{Gate}(g) \rightarrow \text{Circuit}(g).$$

5. Encode a description of the problem instance:

- Now we encode problem of circuit C1, firstly we categorize the circuit and its gate components. This step is easy if ontology about the problem is already thought.
- This step involves the writing simple atomics sentences of instances of concepts, which is known as ontology.
- For the given circuit C1, we can encode the problem instance in atomic sentences as below:
- Since in the circuit there are two XOR, two AND, and one OR gate so atomic sentences for these gates will be:

For XOR gate: Type(x1)= XOR, Type(X2) = XOR

For AND gate: Type(A1) = AND, Type(A2)= AND

For OR gate: Type (O1) = OR.

And then represent the connections between all the gates.

6. Pose queries to the inference procedure and get answers:

In this step, we will find all the possible set of values of the entire terminal for the adder circuit. The first query will be:

What should be the combination of input which would generate the first output of circuit C1, as 0 and a second output to be 1?

$$\exists\ i1, i2, i3\ \text{Signal}(\text{In}(1, C1))=i1 \wedge \text{Signal}(\text{In}(2, C1))=i2 \wedge \text{Signal}(\text{In}(3, C1))= i3$$
$$\wedge\ \text{Signal}(\text{Out}(1, C1)) =0 \wedge \text{Signal}(\text{Out}(2, C1))=1$$

7. Debug the knowledge base:

Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of knowledge base. In the knowledge base, we may have omitted assertions like $1 \neq 0$.

Inference in First-Order Logic

Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences. Before understanding the FOL inference rule, let's understand some basic terminologies used in FOL.

(a)Substitution:

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write **F[a/x]**, so it refers to substitute a constant "**a**" in place of variable "**x**".

Note: First-order logic is capable of expressing facts about some or all objects in the universe.

(b)Equality:

First-Order logic does not only use predicate and terms for making atomic sentences but also uses another way, which is equality in FOL. For this, we can use **equality symbols** which specify that the two terms refer to the same object.

Example: Brother (John) = Smith.

As in the above example, the object referred by the **Brother (John)** is similar to the object referred by **Smith**. The equality symbol can also be used with negation to represent that two terms are not the same objects.

Example: ¬(x=y) which is equivalent to x ≠y.

FOL inference rules for quantifier:

As propositional logic we also have inference rules in first-order logic, so following are some basic inference rules in FOL:

- **Universal Generalization**
- **Universal Instantiation**
- **Existential Instantiation**
- **Existential introduction**

1. Universal Generalization:

- Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).
- It can be represented as: $\frac{P(c)}{\forall x\, P(x)}$.
- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.

Example: Let's represent, P(c): "**A byte contains 8 bits**", so for **∀ x P(x)** "**All bytes contain 8 bits**.", it will also be true.

2. Universal Instantiation:

- Universal instantiation is also called as universal elimination or UI is a valid inference rule. It can be applied multiple times to add new sentences.
- The new KB is logically equivalent to the previous KB.
- As per UI, **we can infer any sentence obtained by substituting a ground term for the variable**.
- The UI rule state that we can infer any sentence P(c) by substituting a ground term c (a constant within domain x) from **∀ x P(x) for any object in the universe of discourse**.
- It can be represented as: $\frac{\forall x\, P(x)}{P(c)}$.

Example:1.

IF "Every person like ice-cream"=> ∀x P(x) so we can infer that
"John likes ice-cream" => P(c)

Example: 2.

Let's take a famous example,

"All kings who are greedy are Evil." So let our knowledge base contains this detail as in the form of FOL:

∀x king(x) ∧ greedy (x) → Evil (x),

So from this information, we can infer any of the following statements using Universal Instantiation:

- **King(John) ∧ Greedy (John) → Evil (John),**
- **King(Richard) ∧ Greedy (Richard) → Evil (Richard),**
- **King(Father(John)) ∧ Greedy (Father(John)) → Evil (Father(John)),**

3. Existential Instantiation:

- Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.
- It can be applied only once to replace the existential sentence.
- The new KB is not logically equivalent to old KB, but it will be satisfiable if old KB was satisfiable.
- This rule states that one can infer P(c) from the formula given in the form of ∃x P(x) for a new constant symbol c.
- The restriction with this rule is that c used in the rule must be a new term for which P(c) is true.
- It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Example:

From the given sentence: **∃x Crown(x) ∧ OnHead(x, John),**

So we can infer: **Crown(K) ∧ OnHead(K, John),** as long as K does not appear in the knowledge base.

- The above used K is a constant symbol, which is called **Skolem constant**.
- The Existential instantiation is a special case of **Skolemization process**.

4. Existential introduction

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.

- It can be represented as: $\frac{P(c)}{\exists x P(x)}$
- **Example: Let's say that,**

  "Priyanka got good marks in English."

  "Therefore, someone got good marks in English."

Generalized Modus Ponens Rule:

For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens.

Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."

According to Modus Ponens, for atomic sentences **pi, pi', q**. Where there is a substitution θ such that SUBST **(θ, pi',) = SUBST(θ, pi)**, it can be represented as:

$$\frac{p1', p2', \ldots, pn', (p1 \wedge p2 \wedge \ldots \wedge pn \Rightarrow q)}{SUBST(\theta, q)}$$

Example:

We will use this rule for Kings are evil, so we will find some x such that x is king, and x is greedy so we can infer that x is evil.

1. Here let say, p1' is king(John) p1 is king(x)
2. p2' is Greedy(y) p2 is Greedy(x)
3. θ is {x/John, y/John} q is evil(x)
4. SUBST(θ,q).

Unification:

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\boldsymbol{\Psi_1\sigma = \Psi_2\sigma}$, then it can be expressed as **UNIFY(Ψ_1, Ψ_2)**.
- **Example: Find the MGU for Unify{King(x), King(John)}**

Let Ψ_1 = King(x), Ψ_2 = King(John),

Substitution θ = {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).
- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

E.g. Let's say there are two different expressions, **P(x, y), and P(a, f(z))**.

In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

P(x, y)..... (i)
P(a, f(z))..... (ii)

- Substitute x with a, and y with f(z) in the first expression, and it will be represented as **a/x** and f(z)/y.
- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: **[a/x, f(z)/y]**.

Conditions for Unification:

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm:

Algorithm: Unify(Ψ_1, Ψ_2)

Step. 1: If Ψ_1 or Ψ_2 is a variable or constant, then:
  a) If Ψ_1 or Ψ_2 are identical, then return NIL.
  b) Else if $\Psi_1$is a variable,
    a. then if Ψ_1 occurs in Ψ_2, then return FAILURE
    b. Else return { (Ψ_2/ Ψ_1)}.
  c) Else if Ψ_2 is a variable,
    a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
    b. Else return {(Ψ_1/ Ψ_2)}.
  d) Else return FAILURE.
Step.2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.
Step. 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.
Step. 4: Set Substitution set(SUBST) to NIL.
Step. 5: For i=1 to the number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

Implementation of the Algorithm

Step.1: Initialize the substitution set to be empty.

Step.2: Recursively unify atomic sentences:

a. Check for Identical expression match.

b. If one expression is a variable v_i, and the other is a term t_i which does not contain variable v_i, then:

a. Substitute t_i / v_i in the existing substitutions

b. Add t_i /v_i to the substitution setlist.

c. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

For each pair of the following atomic sentences find the most general unifier (If exist).

1. Find the MGU of {p(f(a), g(Y)) and p(X, X)}

Sol: S_0 => Here, Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(X, X)
SUBST θ= {f(a) / X}
S1 => Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(f(a), f(a))
SUBST θ= {f(a) / g(y)}, **Unification failed**.

Unification is not possible for these expressions.

2. Find the MGU of {p(b, X, f(g(Z))) and p(Z, f(Y), f(Y))}

Here, Ψ_1 = p(b, X, f(g(Z))) , and Ψ_2 = p(Z, f(Y), f(Y))
S_0 => { p(b, X, f(g(Z))); p(Z, f(Y), f(Y))}
SUBST θ={b/Z}

S_1 => { p(b, X, f(g(b))); p(b, f(Y), f(Y))}
SUBST θ={f(Y) /X}

S_2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))}
SUBST θ= {g(b) /Y}

S_2 => { p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))} **Unified Successfully.**
And Unifier = { b/Z, f(Y) /X , g(b) /Y}.

3. Find the MGU of {p (X, X), and p (Z, f(Z))}

Here, Ψ_1 = {p (X, X), and Ψ_2 = p (Z, f(Z))
S_0 => {p (X, X), p (Z, f(Z))}
SUBST θ= {X/Z}
S1 => {p (Z, Z), p (Z, f(Z))}
SUBST θ= {f(Z) / Z}, **Unification Failed**.

Hence, unification is not possible for these expressions.

4. Find the MGU of UNIFY(prime (11), prime(y))

Here, Ψ_1 = {prime(11) , and Ψ_2 = prime(y)}
S_0 => {prime(11) , prime(y)}
SUBST θ= {11/y}

S_1 => {prime(11) , prime(11)} , **Successfully unified.**
Unifier: {11/y}.

5. Find the MGU of Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)}

Here, Ψ_1 = Q(a, g(x, a), f(y)), and Ψ_2 = Q(a, g(f(b), a), x)
S_0 => {Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)}
SUBST θ= {f(b)/x}
S_1 => {Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))}

SUBST θ= {b/y}
S_1 => {Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))}, **Successfully Unified.**

Unifier: [a/a, f(b)/x, b/y].

6. UNIFY(knows(Richard, x), knows(Richard, John))

Here, Ψ_1 = knows(Richard, x), and Ψ_2 = knows(Richard, John)
S_0 => { knows(Richard, x); knows(Richard, John)}
SUBST θ= {John/x}
S_1 => { knows(Richard, John); knows(Richard, John)}, **Successfully Unified.**
Unifier: {John/x}.

Resolution in FOL

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. It was invented by a Mathematician John Alan Robinson in the year 1965.

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form**.

Clause: Disjunction of literals (an atomic sentence) is called a **clause**. It is also known as a unit clause.

Conjunctive Normal Form: A sentence represented as a conjunction of clauses is said to be **conjunctive normal form** or **CNF**.

Note: To better understand this topic, firstly learns the FOL in AI.

The resolution inference rule:

The resolution rule for first-order logic is simply a lifted version of the propositional rule. Resolution can resolve two clauses if they contain complementary literals, which are assumed to be standardized apart so that they share no variables.

$$\frac{l_1 \vee \ldots\ldots \vee l_{k,} \qquad m_1 \vee \ldots\ldots \vee m_n}{\text{SUBST}(\theta, l_1 \vee \ldots\ldots\ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots\ldots \vee l_k \vee m_1 \vee \ldots\ldots\ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots\ldots \vee m_n)}$$

Where $\mathbf{l_i}$ and $\mathbf{m_j}$ are complementary literals.

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

Example:

We can resolve two clauses which are given below:

[Animal (g(x) V Loves (f(x), x)] **and** **[¬ Loves(a, b) V ¬Kills(a, b)]**

Where two complimentary literals are: **Loves (f(x), x) and ¬ Loves (a, b)**

These literals can be unified with unifier **θ= [a/f(x), and b/x]** , and it will generate a resolvent clause:

[Animal (g(x) V ¬ Kills(f(x), x)].

Steps for Resolution:

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

To better understand all the above steps, we will take an example in which we will apply resolution.

Example:

a. **John likes all kind of food.**

b. **Apple and vegetable are food**

c. **Anything anyone eats and not killed is food.**

d. **Anil eats peanuts and still alive**

e. **Harry eats everything that Anil eats.**

Prove by resolution that:

f. **John likes peanuts.**

Step-1: Conversion of Facts into FOL

In the first step we will convert all the given statements into its first order logic.

a. $\forall x$: food(x) → likes(John, x)

b. food(Apple) Λ food(vegetables)

c. $\forall x \forall y$: eats(x, y) Λ ¬ killed(x) → food(y)

d. eats (Anil, Peanuts) Λ alive(Anil).

e. $\forall x$: eats(Anil, x) → eats(Harry, x)

f. $\forall x$: ¬ killed(x) → alive(x) } **added predicates.**

g. $\forall x$: alive(x) →¬ killed(x) }

h. likes(John, Peanuts)

Step-2: Conversion of FOL into CNF

In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.

- **Eliminate all implication (→) and rewrite**
    a. $\forall x$ ¬ food(x) V likes(John, x)
    b. food(Apple) Λ food(vegetables)
    c. $\forall x \forall y$ ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
    d. eats (Anil, Peanuts) Λ alive(Anil)
    e. $\forall x$ ¬ eats(Anil, x) V eats(Harry, x)
    f. $\forall x$¬ [¬ killed(x)] V alive(x)
    g. $\forall x$ ¬ alive(x) V ¬ killed(x)
    h. likes(John, Peanuts).
- **Move negation (¬)inwards and rewrite**
    a. $\forall x$ ¬ food(x) V likes(John, x)
    b. food(Apple) Λ food(vegetables)
    c. $\forall x \forall y$ ¬ eats(x, y) V killed(x) V food(y)
    d. eats (Anil, Peanuts) Λ alive(Anil)
    e. $\forall x$ ¬ eats(Anil, x) V eats(Harry, x)
    f. $\forall x$ ¬killed(x)] V alive(x)

g. ∀x ¬ alive(x) V ¬ killed(x)

h. likes(John, Peanuts).

- **Rename variables or standardize variables**

  a. ∀x ¬ food(x) V likes(John, x)

  b. food(Apple) Λ food(vegetables)

  c. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)

  d. eats (Anil, Peanuts) Λ alive(Anil)

  e. ∀w¬ eats(Anil, w) V eats(Harry, w)

  f. ∀g ¬killed(g)] V alive(g)

  g. ∀k ¬ alive(k) V ¬ killed(k)

  h. likes(John, Peanuts).

- **Eliminate existential instantiation quantifier by elimination.**

  In this step, we will eliminate existential quantifier ∃, and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.

- **Drop Universal quantifiers.**

  In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.

  a. ¬ food(x) V likes(John, x)

  b. food(Apple)

  c. food(vegetables)

  d. ¬ eats(y, z) V killed(y) V food(z)

  e. eats (Anil, Peanuts)

  f. alive(Anil)

  g. ¬ eats(Anil, w) V eats(Harry, w)

  h. killed(g) V alive(g)

  i. ¬ alive(k) V ¬ killed(k)

  j. likes(John, Peanuts).

Note: Statements "food(Apple) Λ food(vegetables)" and "eats (Anil, Peanuts) Λ alive(Anil)" can be written in two separate statements.

- **Distribute conjunction ∧ over disjunction ¬.**

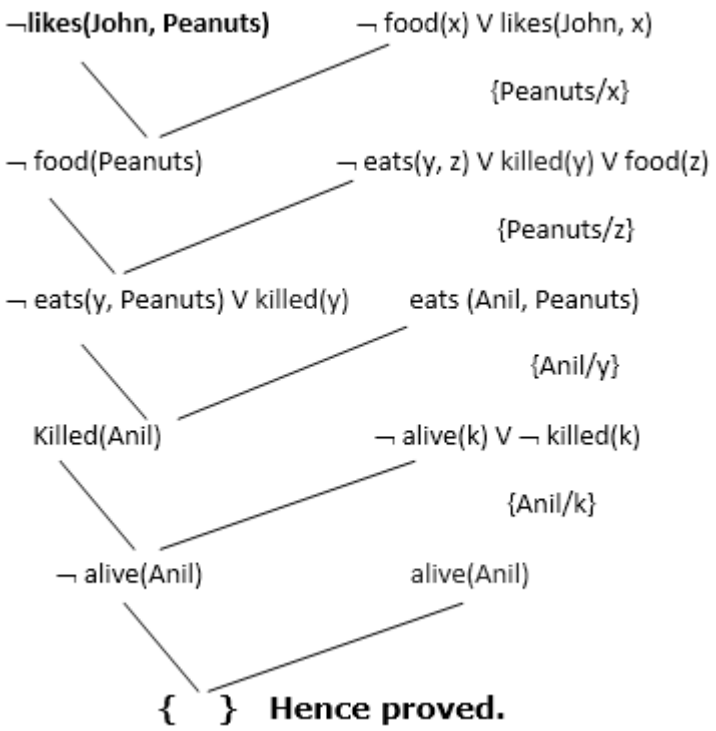
  This step will not make any change in this problem.

Step-3: Negate the statement to be proved

In this statement, we will apply negation to the conclusion statements, which will be written as ¬likes(John, Peanuts)

Step-4: Draw Resolution graph:

Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:



Hence the negation of the conclusion has been proved as a complete contradiction with the given set of statements.

Explanation of Resolution graph:

- In the first step of resolution graph, **¬likes(John, Peanuts)** , and **likes(John, x)** get resolved(canceled) by substitution of **{Peanuts/x}**, and we are left with **¬ food(Peanuts)**
- In the second step of the resolution graph, **¬ food(Peanuts)** , and **food(z)** get resolved (canceled) by substitution of **{ Peanuts/z}**, and we are left with **¬ eats(y, Peanuts) V killed(y)** .
- In the third step of the resolution graph, **¬ eats(y, Peanuts)** and **eats (Anil, Peanuts)** get resolved by substitution **{Anil/y}**, and we are left with **Killed(Anil)** .
- In the fourth step of the resolution graph, **Killed(Anil)** and **¬ killed(k)** get resolve by substitution **{Anil/k}**, and we are left with **¬ alive(Anil)** .
- In the last step of the resolution graph **¬ alive(Anil)** and **alive(Anil)** get resolved.

Forward Chaining and backward chaining in AI

In artificial intelligence, forward and backward chaining is one of the important topics, but before understanding forward and backward chaining lets first understand that from where these two terms came.

Inference engine:

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts. The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:

a. **Forward chaining**

b. **Backward chaining**

Horn Clause and Definite clause:

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the **first-order definite clause**.

Definite clause: A clause which is a disjunction of literals with **exactly one positive literal** is known as a definite clause or strict horn clause.

Horn clause: A clause which is a disjunction of literals with **at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.

Example: (¬ p V ¬ q V k). It has only one positive literal k.

It is equivalent to p ∧ q → k.

(a) Forward Chaining

Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.

The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

Properties of Forward-Chaining:

- It is a down-up approach, as it moves from bottom to top.
- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.
- Forward-chaining approach is also called as data-driven as we reach to the goal using available data.

- Forward -chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Consider the following famous example which we will use in both approaches:

Example:

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen."

Prove that **"Robert is criminal."**

To solve the above problem, first, we will convert all the above facts into first-order definite clauses, and then we will use a forward-chaining algorithm to reach the goal.

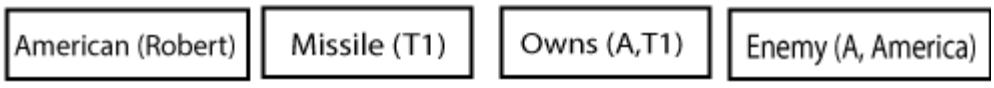
Facts Conversion into FOL:

- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
  American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)
- Country A has some missiles. **?p Owns(A, p) ∧ Missile(p)**. It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.
  Owns(A, T1) .....(2)
  Missile(T1) .....(3)
- All of the missiles were sold to country A by Robert.
  ?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) .....(4)
- Missiles are weapons.
  Missile(p) → Weapons (p) .....(5)
- Enemy of America is known as hostile.
  Enemy(p, America) →Hostile(p) .....(6)
- Country A is an enemy of America.
  Enemy (A, America) .....(7)
- Robert is American
  American(Robert). .....(8)

Forward chaining proof:

Step-1:

In the first step we will start with the known facts and will choose the sentences which do not have implications, such as: **American(Robert), Enemy(A, America), Owns(A, T1), and Missile(T1)**. All these facts will be represented as below.



Step-2:

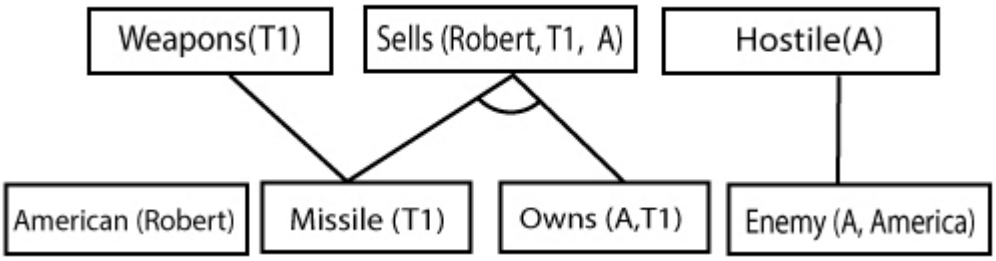
At the second step, we will see those facts which infer from available facts and with satisfied premises.

Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

Rule-(2) and (3) are already added.

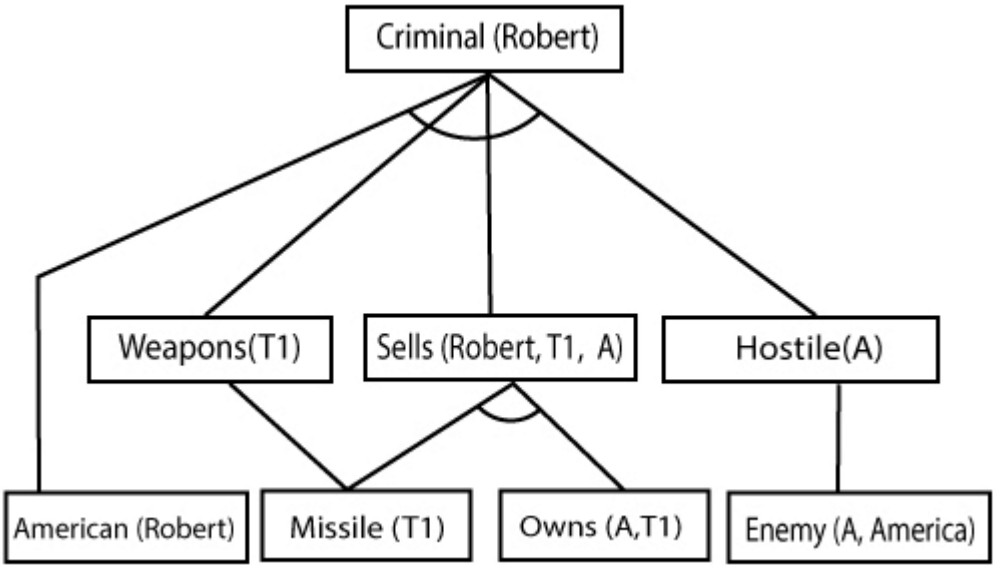
Rule-(4) satisfy with the substitution {p/T1}, **so Sells (Robert, T1, A)** is added, which infers from the conjunction of Rule (2) and (3).

Rule-(6) is satisfied with the substitution(p/A), so Hostile(A) is added and which infers from Rule-(7).



Step-3:

At step-3, as we can check Rule-(1) is satisfied with the substitution **{p/Robert, q/T1, r/A}, so we can add Criminal(Robert)** which infers all the available facts. And hence we reached our goal statement.



Hence it is proved that Robert is Criminal using forward chaining approach.

(b) Backward Chaining:

Backward-chaining is also known as a backward deduction or backward reasoning method when using an inference engine. A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

Properties of backward chaining:

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a **depth-first search** strategy for proof.

Example:

In backward-chaining, we will use the same above example, and will rewrite all the rules.

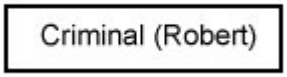
- **American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)**
  Owns(A, T1) .....(2)
- **Missile(T1)**
- **?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) .....(4)**
- **Missile(p) → Weapons (p) .....(5)**
- **Enemy(p, America) →Hostile(p) .....(6)**
- **Enemy (A, America) .....(7)**
- **American(Robert). .....(8)**

Backward-Chaining proof:

In Backward chaining, we will start with our goal predicate, which is **Criminal(Robert)**, and then infer further rules.

Step-1:

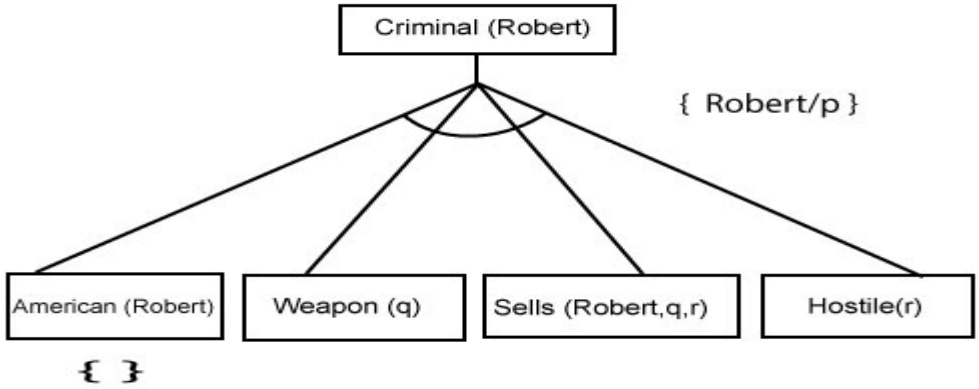
At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.



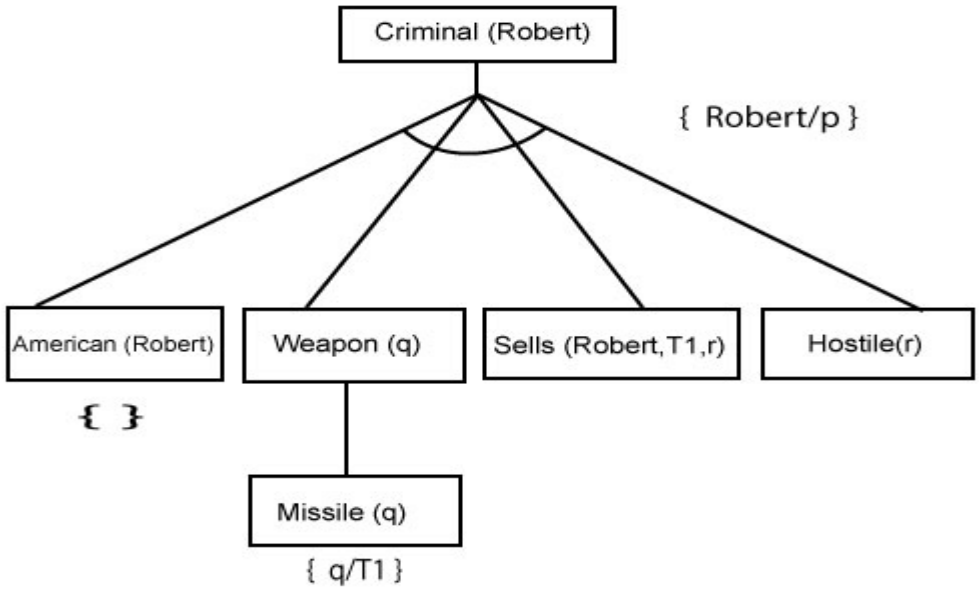
Step-2:

At the second step, we will infer other facts form goal fact which satisfies the rules. So as we can see in Rule-1, the goal predicate Criminal (Robert) is present with substitution {Robert/P}. So we will add all the conjunctive facts below the first level and will replace p with Robert.

Here we can see American (Robert) is a fact, so it is proved here.

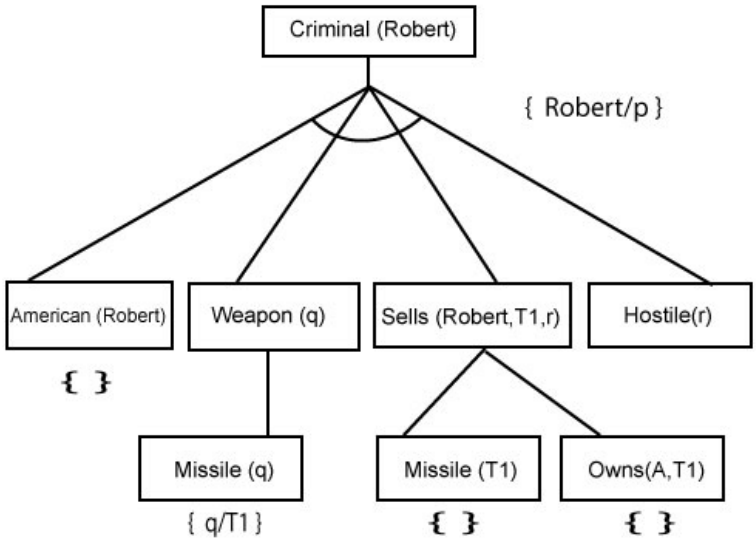


Step-3:t At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.



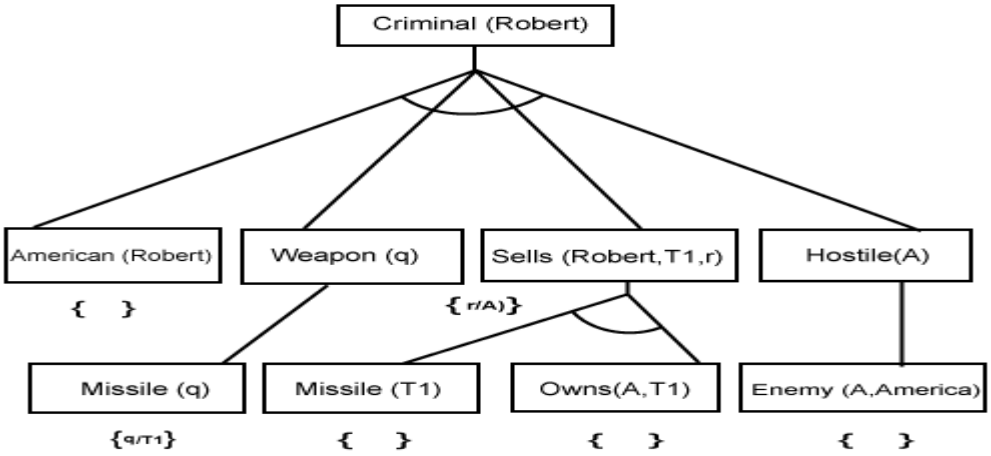
Step-4:

At step-4, we can infer facts Missile(T1) and Owns(A, T1) form Sells(Robert, T1, r) which satisfies the **Rule- 4**, with the substitution of A in place of r. So these two statements are proved here.



Step-5:

At step-5, we can infer the fact **Enemy(A, America)** from **Hostile(A)** which satisfies Rule- 6. And hence all the statements are proved true using backward chaining.



KNOWLEDGE REPRESENTATION

ONTOLOGICAL ENGINEERING

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

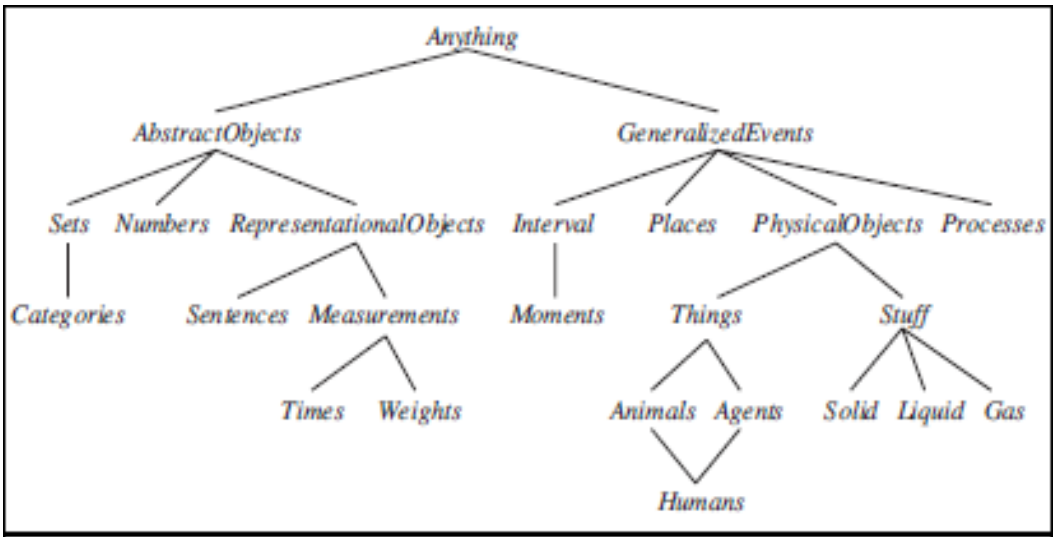


Figure 12.1 The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint; a human is both an animal and an agent, for example. We will see in Section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

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

Categories and Objects

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

rather than a *particular* basketball such as BB9

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

Balls), abbreviated as Basketballs ⊂ Balls, to say that Basketballs is a **subcategory** of Balls.

Categories serve to organize and simplify the knowledge base through **inheritance**. If we say that all instances of the category Food are edible, and if we assert that Fruit is a subclass of Food and Apples is a subclass of Fruit , then we can infer that every apple is edible. We say that the individual apples **inherit** the property of edibility, in this case from their membership in the Food category.
First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members. Here are some types of facts, with examples of each:

· An object is a member of a category.
BB9 ∈ *Basketballs*

· A category is a subclass of another category.
Basketballs ⊂ *Balls*

· All members of a category have some properties.
*(*x∈ *Basketballs)* ⇒ *Spherical (*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*

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

Categories can also be *defined* by providing necessary and sufficient conditions for membership. For example, a bachelor is an unmarried adult male:
x∈ *Bachelors* ⇔ *Unmarried(*x*)* ∧ x∈ *Adults* ∧ x∈*Males*

Physical Composition

We use the general PartOf relation to say that one thing is part of another. Objects can be grouped into part of hierarchies, reminiscent of the Subset hierarchy:
PartOf (Bucharest , *Romania) PartOf (Romania*,*EasternEurope) PartOf (EasternEurope*, *Europe)*

PartOf (Europe, *Earth)*

The PartOf relation is transitive and reflexive; that is, *PartOf (*x, y*)* ∧ *PartOf (*y, z*)* ⇒ *PartOf (*x, z*) PartOf (*x, x*)*

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

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

BunchOf ({Apple1,Apple2,Apple3})

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

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

∀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*)*

Measurements

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

*Length(*L*1)=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*)*

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

Diameter (Basketball12)=Inches(9.5)

ListPrice(Basketball12)=$(19)

d∈ *Days* ⇒ *Duration(*d*)=Hours(24)*

Time Intervals

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

Partition({Moments,ExtendedIntervals},Intervals)

i∈*Moments* ⇔ *Duration(*i*)=Seconds(0)*

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

*Interval (*i*) ⇒ Duration(*i*)=(Time(End(*i*)) − Time(Begin(*i*))) Time(Begin(AD1900))=Seconds(0)*
Time(Begin(AD2001))=Seconds(3187324800) Time(End(AD2001))=Seconds(3218860800)

Duration(AD2001)=Seconds(31536000)

Two intervals Meet if the end time of the first equals the start time of the second. The complete set of interval relations, as proposed by Allen (1983), is shown graphically in Figure 12.2 and logically below:
*Meet(*i,j*) ⇔ End(*i*)=Begin(*j*) Before(*i,j*) ⇔ End(*i*) <Begin(*j*) After (*j,i*) ⇔ Before(*i, j*)*

*During(*i,j*) ⇔ Begin(*j*) < Begin(*i*) < End(*i*) < End(*j*) Overlap(*i,j*) ⇔ Begin(*i*) < Begin(*j*) < End(*i*) < End(*j*)*
*Begins(*i,j*) ⇔ Begin(*i*) = Begin(*j*)*

*Finishes(*i,j*) ⇔ End(*i*) = End(*j*)*

*Equals(*i,j*) ⇔ Begin(*i*) = Begin(*j*) ∧ End(*i*) = End(*j*)*

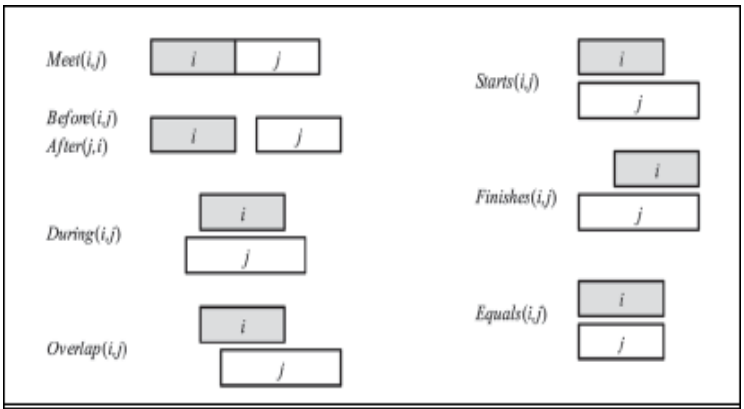


Figure 12.2 Predicates on time intervals.

EVENTS

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

Events are described as instances of event categories. The event E1 of Shankar flying from San Francisco to Washington, D.C. is described as
E*1* ∈ Flyings ∧ *Flyer (*E*1*, *Shankar) ∧ Origin(*E*1*, *SF) ∧ Destination(*E*1*, *DC)* we can define an alternative three-argument version of the category of flying events and say E*1* ∈ Flyings*(Shankar , SF*,*DC)*

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

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

*Initiates(*e, f, t*)* Event e causes fluent f to start to hold at time t *Terminates(*e, f, t*)* Event e causes fluent f to cease to hold at time t *Clipped(*f, i*)* Fluent f ceases to be true at some point during time interval i *Restored (*f, i*)* Fluent f becomes true sometime during time interval i

We assume a distinguished event, Start , that describes the initial state by saying which fluents are initiated or terminated at the start time. We define T by saying that a fluent holds at a point in time if the fluent was initiated by an event at some time in the past and was not made false (clipped) by an intervening event. A fluent does not hold if it was terminated by an event and not made true (restored) by another event. Formally, the axioms are:
$Happens(e, (t1, t2)) \wedge Initiates(e, f, t1) \wedge \neg Clipped(f, (t1, t)) \wedge t1 < t \Rightarrow T(f, t)$ $Happens(e, (t1, t2)) \wedge Terminates(e, f, t1) \wedge \neg Restored(f, (t1, t)) \wedge t1 < t \Rightarrow \neg T(f, t)$ where Clipped and Restored are defined by

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

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

MENTAL EVENTS AND MENTAL OBJECTS

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

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

Knows(Lois, CanFly(Superman))

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

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

In first-order logic a **model** contains a set of objects and an interpretation that maps each name to the appropriate object, relation, or function. In modal logic we want to be able to consider both the possibility that Superman's secret identity is Clark and that it isn't. Therefore, we will need a more complicated model, one that consists of a collection of **possible worlds** rather than just one true world. The worlds are connected in a graph by **accessibility relations**, one relation for each modal operator.

We say that world w1 is accessible from world w0 with respect to the modal operator **K**A if everything in w1 is consistent with what A knows in w0, and we write this as Acc(**K**A,w0,w1). In diagrams such as Figure 12.4 we show accessibility as an arrow between possible worlds.

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

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

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

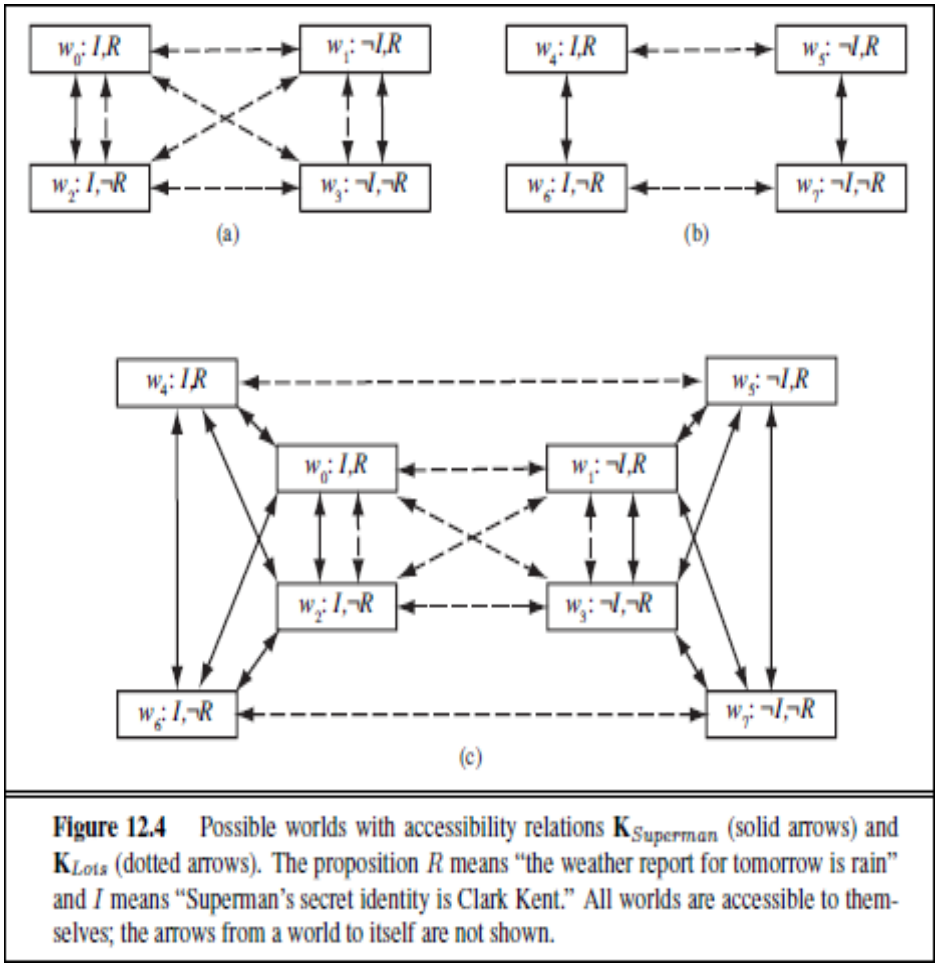


Figure 12.4 Possible worlds with accessibility relations $\mathbf{K}_{Superman}$ (solid arrows) and $\mathbf{K}_{Lois}$ (dotted arrows). The proposition *R* means "the weather report for tomorrow is rain" and *I* means "Superman's secret identity is Clark Kent." All worlds are accessible to themselves;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

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

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

In the BOTTOM diagram we represent the scenario where it is common knowledge that Superman knows his identity, and Lois might or might not have seen the weather report. We represent this by combining the two top scenarios, and adding arrows to show that Superman does not know

which scenario actually holds. Lois does know, so we don't need to add any arrows for her. In w0 Superman still knows I but not R, and now he does not know whether Lois knows R. From what Superman knows, he might be in w0 or w2, in which case Lois does not know whether R is true, or he could be in w4, in which case she knows R, or w6, in which case she knows ¬R.

REASONING SYSTEMS FOR CATEGORIES

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

SEMANTIC NETWORKS

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

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

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

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

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

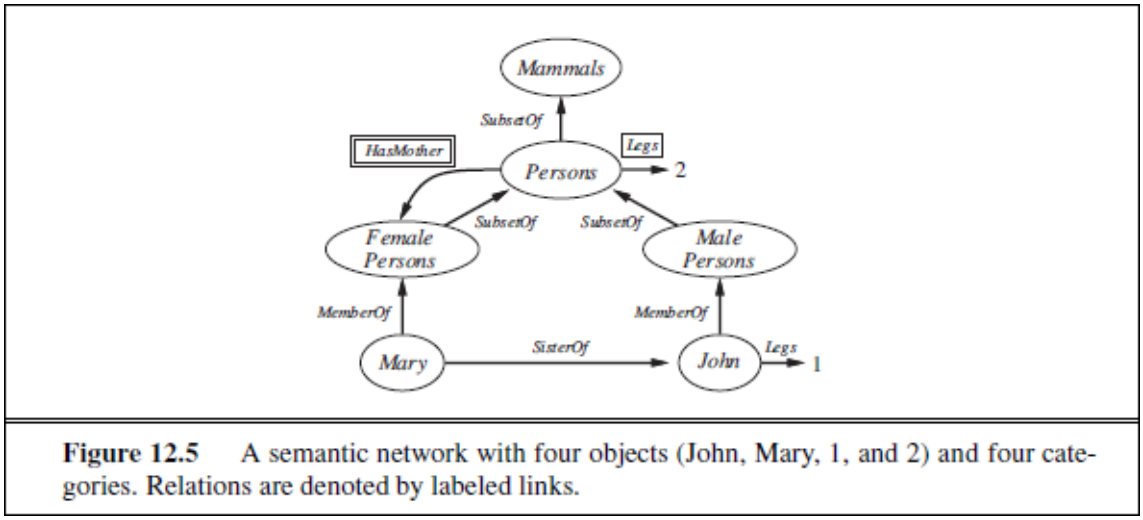


Figure 12.5 A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

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

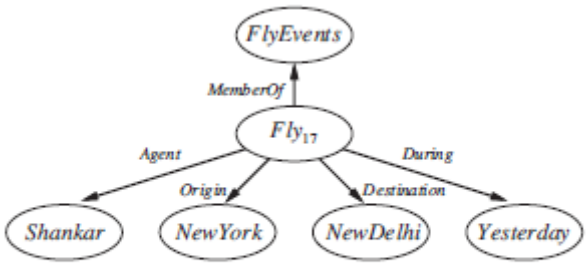


Figure 12.6 A fragment of a semantic network showing the representation of the logical assertion $Fly(Shankar, NewYork, NewDelhi, Yesterday)$.

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

3.1.1.1 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.

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** 3 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) A Adult(x) A Male(x)

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

The description logic effectively allows direct logical operations on predicates, rather than sentences joined by connectives.

Figure 10.11 Syntax of CLASSIC Language

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.

Open and Closed Worlds

If a university advertises 5Courses offered are CS01, CS02, CS03 and EE01", the answer for the question 5How many courses are offered by the university?" is 5four. But in first order logic the answer is 5at 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.

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 A β |= α. Under the closed-world assumption, if a proposition α is not mentioned in KB then KB |= ¬α, but KB A α |= α.

These failures of monotonicity are widespread in common sense reasoning. Because humans often 5jump 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.**

Circumscription

The idea is to specify particular predicates that are assumed to be 5as false as possible"4 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

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

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 5Nixon 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) A Quaker(Nixon) . Republican(x) A¬Abnormal 2(x) ⇒ ¬Pacifist (x) .

Quaker(x) A ¬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.

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) A Quaker(Nixon) .

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

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

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 ⇒ Q}, it would be removed; if it had the additional justification {P, P v R ⇒ Q}, it would still be removed; but if it also had the justification {R, P vR ⇒ 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.