UNIT-3

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.

First-Order Logic

n 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:
    a. Objects: A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, ...

b. 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

c. 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

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.

Basic Elements of First-order logic:

Following are the 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	∧, ∨, ¬, ⇒, ⇔
Equality	==
Quantifier	∀, ∃

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).

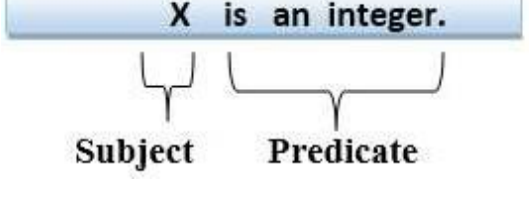
Complex Sentences:

- 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).

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 $\forall$, which resembles an inverted A.

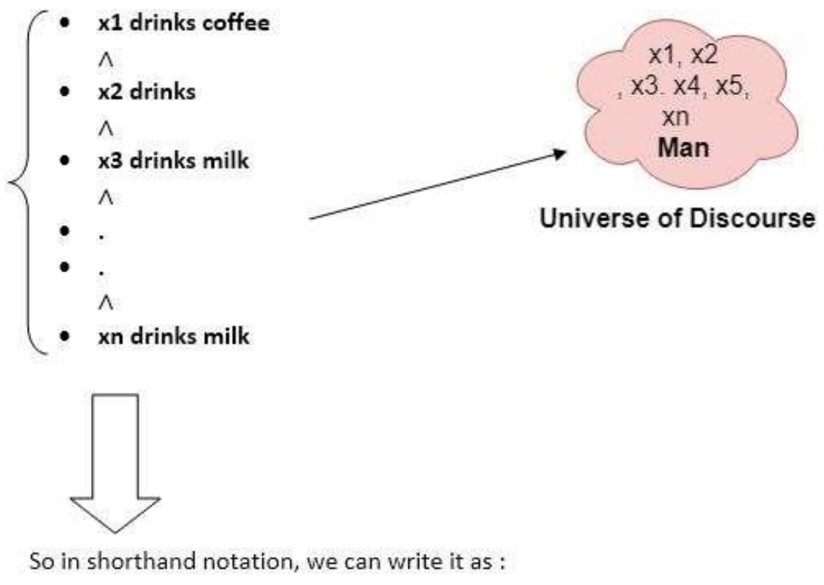
If x is a variable, then $\forall x$ is read as:

- For all x
- For each x
- For every x.

Example:

All man drink coffee.

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



∀x man(x) → drink (x, coffee).

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

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.

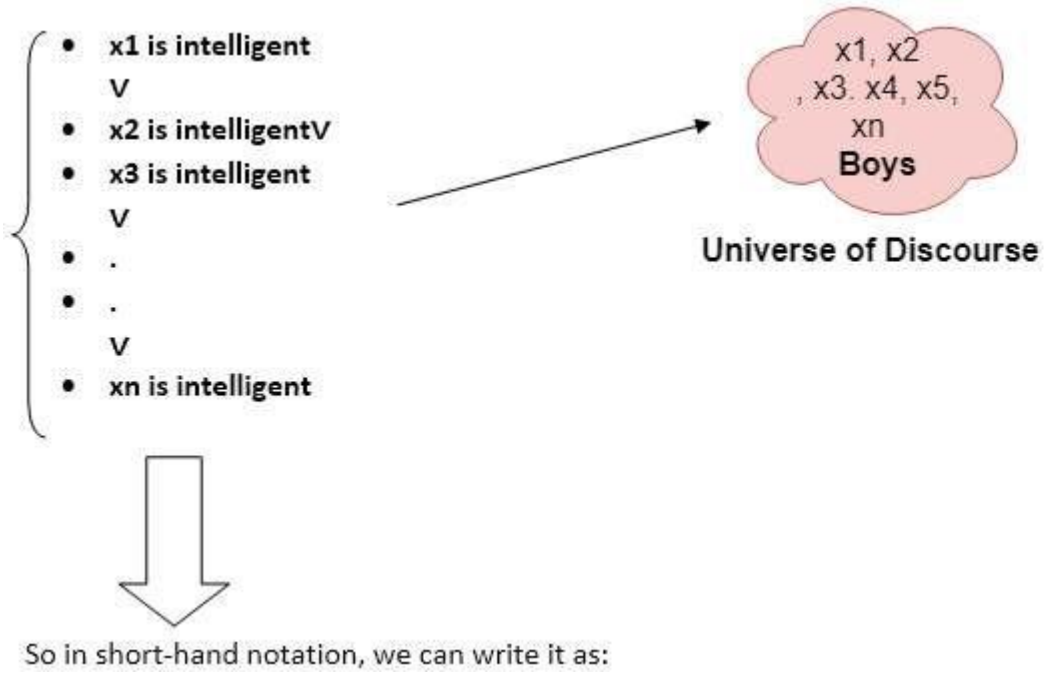
It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

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.'

Example:

Some boys are intelligent.



∃x: boys(x) ∧ intelligent(x)

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

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:

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

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:

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

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: ∀x ∃(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: ∀x [A (x) B(y)], here x and y are the bound variables.

Knowledge Engineering in First-order logic

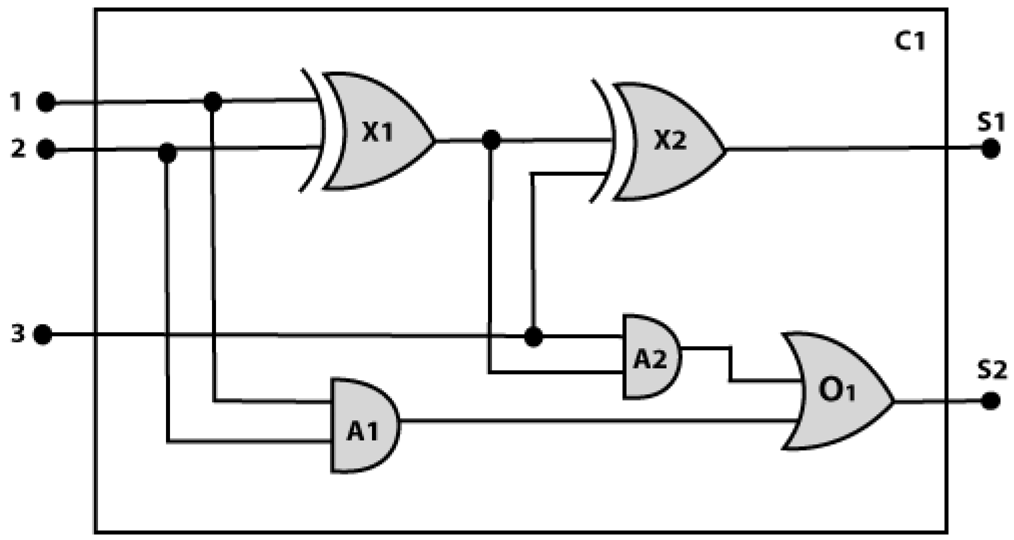
What is knowledge-engineering?

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:

1. $\forall$ t1, t2 Terminal (t1) $\land$ Terminal (t2) $\land$ Connect (t1, t2) $\rightarrow$ Signal (t1) = Signal (2).

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

1. $\forall$ t Terminal (t) $\rightarrow$Signal (t) = 1 $\lor$Signal (t) = 0.

- Connect predicates are commutative:

1. $\forall$ t1, t2 Connect(t1, t2) $\rightarrow$ Connect (t2, t1).

- Representation of types of gates:

1. $\forall$ g Gate(g) $\land$ r = Type(g) $\rightarrow$ r = OR $\lor$r = AND $\lor$r = XOR $\lor$r = NOT.

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

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

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

1. $\forall$ g Gate(g) $\land$ Type(g) = OR $\rightarrow$ Signal (Out(1, g))= 1 $\Leftrightarrow$ $\exists$n Signal (In(n, g))= 1

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

1. $\forall$ g Gate(g) $\land$ Type(g) = XOR $\rightarrow$ Signal (Out(1, g)) = 1 $\Leftrightarrow$ Signal (In(1, g)) $\neq$ Signal (In(2, g)).

- Output of NOT gate is invert of its input:

1. $\forall$ g Gate(g) $\land$ Type(g) = NOT $\rightarrow$ Signal (In(1, g)) $\neq$ Signal (Out(1, g)).

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

1. $\forall$ g Gate(g) $\land$ Type(g) = NOT $\rightarrow$ Arity(g, 1, 1)

2. $\forall$ g Gate(g) $\land$ r =Type(g) $\land$ (r= AND $\lor$ r= OR $\lor$ r= XOR) → Arity (g, 2, 1).

- All gates are logic circuits:

1. $\forall$ g Gate(g) → 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:

1. For XOR gate: Type(x1)= XOR, Type(X2) = XOR
2. For AND gate: Type(A1) = AND, Type(A2)= AND
3. 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 all the 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?

1. $\exists$ i1, i2, i3 Signal (In(1, C1))=i1 $\land$ Signal (In(2, C1))=i2 $\land$ Signal (In(3, C1))= i3
2. $\land$ Signal (Out(1, C1)) =0 $\land$ Signal (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.

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".

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: $\neg(x=y)$ which is equivalent to $x \neq y$.

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:

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

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).

What is 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, $\Psi_1\sigma = \Psi_2\sigma$, 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 $\Psi_1$ or $\Psi_2$ is a variable or constant, then:
    a) If $\Psi_1$ or $\Psi_2$ are identical, then return NIL.
    b) Else if $\Psi_1$is a variable,
        a. then if $\Psi_1$ occurs in $\Psi_2$, then return FAILURE
        b. Else return { ($\Psi_2$/ $\Psi_1$)}.
    c) Else if $\Psi_2$ is a variable,
        a. If $\Psi_2$ occurs in $\Psi_1$ then return FAILURE,
        b. Else return {( $\Psi_1$/ $\Psi_2$)}.
    d) Else return FAILURE.
Step.2: If the initial Predicate symbol in $\Psi_1$ and $\Psi_2$ are not same, then return FAILURE.
Step. 3: IF $\Psi_1$ and $\Psi_2$ 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 $\Psi_1$.
    a) Call Unify function with the ith element of $\Psi_1$ and ith element of $\Psi_2$, 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

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.

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.

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

Where l_i and 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

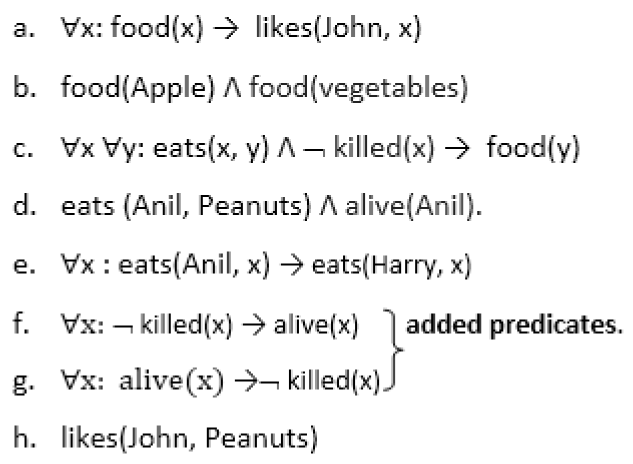
e. Harry eats everything that Anil eats.

   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. ∀x: food(x) → likes(John, x)
b. food(Apple) Λ food(vegetables)
c. ∀x ∀y: eats(x, y) Λ ¬ killed(x) → food(y)
d. eats (Anil, Peanuts) Λ alive(Anil).
e. ∀x : eats(Anil, x) → eats(Harry, x)
f. ∀x: ¬ killed(x) → alive(x)
g. ∀x: alive(x) →¬ killed(x)

added predicates.

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. ∀x ¬ food(x) V likes(John, x)
  b. food(Apple) Λ food(vegetables)
  c. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
  d. eats (Anil, Peanuts) Λ alive(Anil)
  e. ∀x ¬ eats(Anil, x) V eats(Harry, x)
  f. ∀x¬ [¬ killed(x)] V alive(x)

g. ∀x ¬ alive(x) V ¬ killed(x)

h. likes(John, Peanuts).

- Move negation (¬)inwards and rewrite

a. ∀x ¬ food(x) V likes(John, x)

b. food(Apple) Λ food(vegetables)

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

d. eats (Anil, Peanuts) Λ alive(Anil)

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

f. ∀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).

Distribute conjunction ∧ over disjunction ¬.

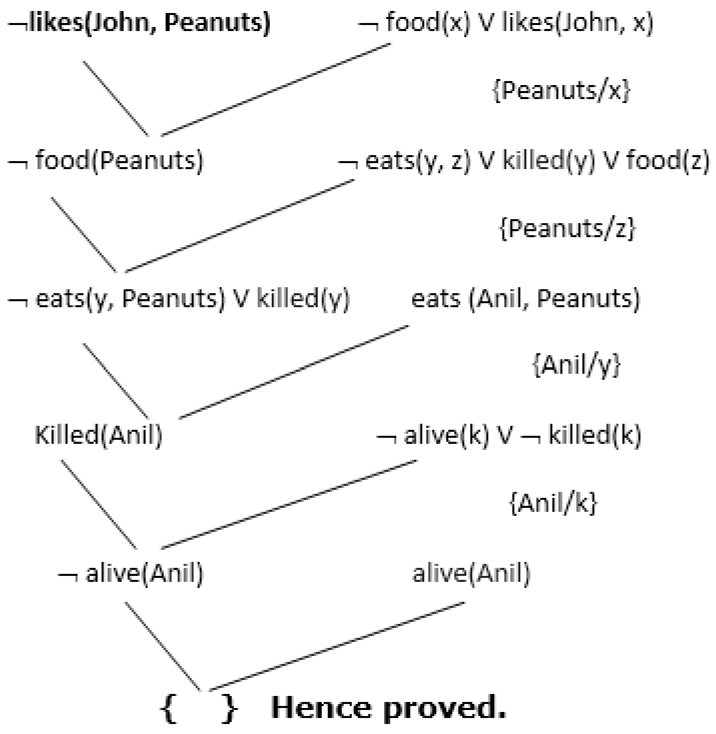
This step will not make any change in this problem.

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: $(\neg p \vee \neg q \vee k)$. It has only one positive literal k.

It is equivalent to $p \wedge q \rightarrow k$.

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:

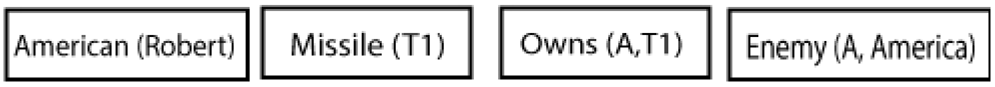
- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
  American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)
- Country A has some missiles. ?p Owns(A, p) ∧ Missile(p). It can be written in two definite clauses by using Existential 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:

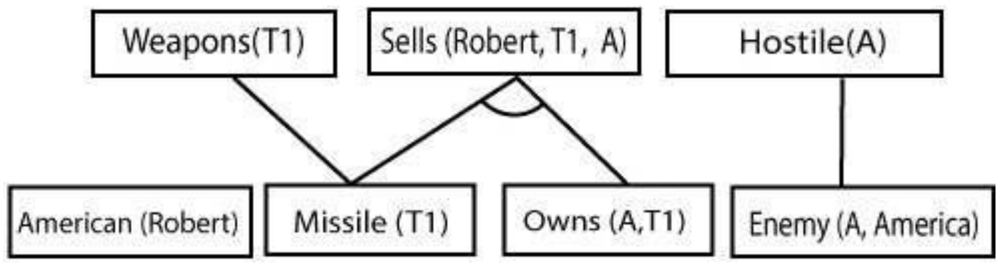
At the second step, we will see those facts which infer from available facts and with satisfied premises.

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

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

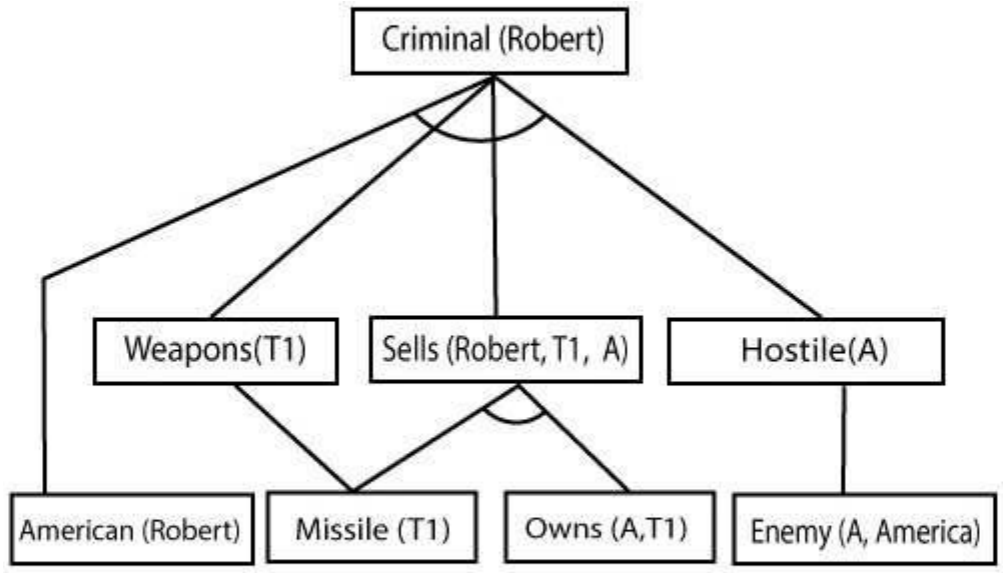
Rule-(4) satisfy with the substitution {p/T1}, so Sells (Robert, T1, A) is added, which infers from the conjunction of Rule (2) and (3).

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)

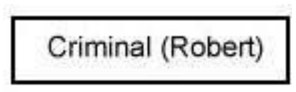
- American(Robert)............................(8)

Backward-Chaining proof:

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

Step-1:

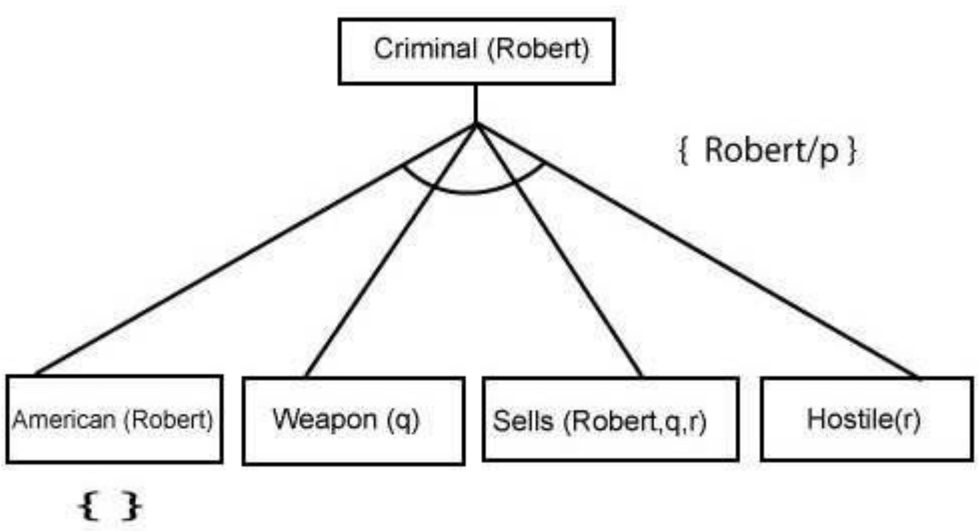
At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.



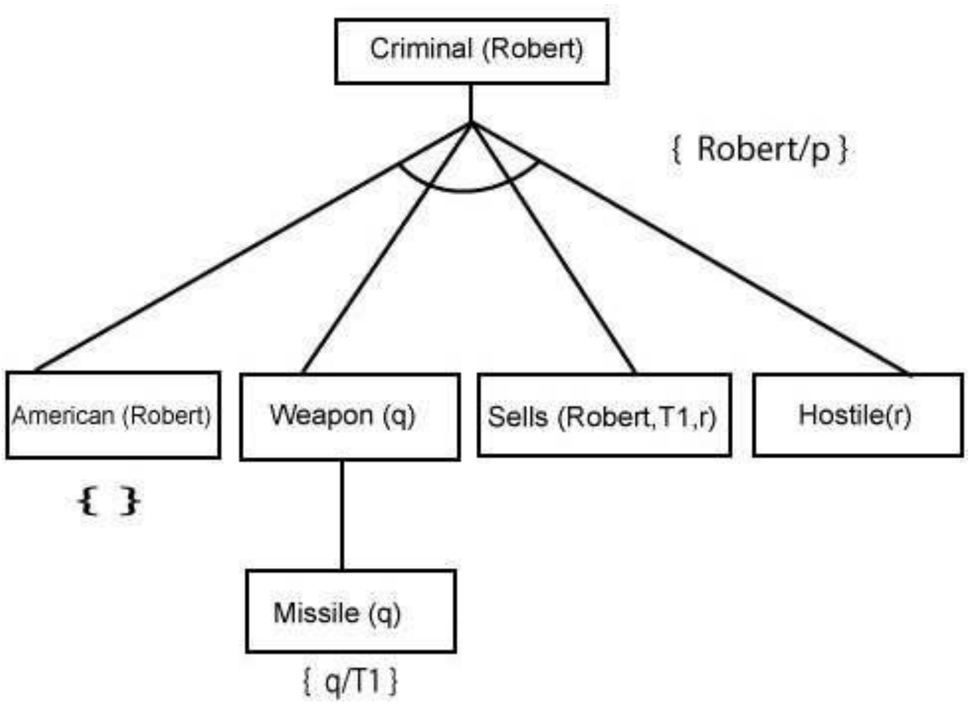
Step-2:

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.

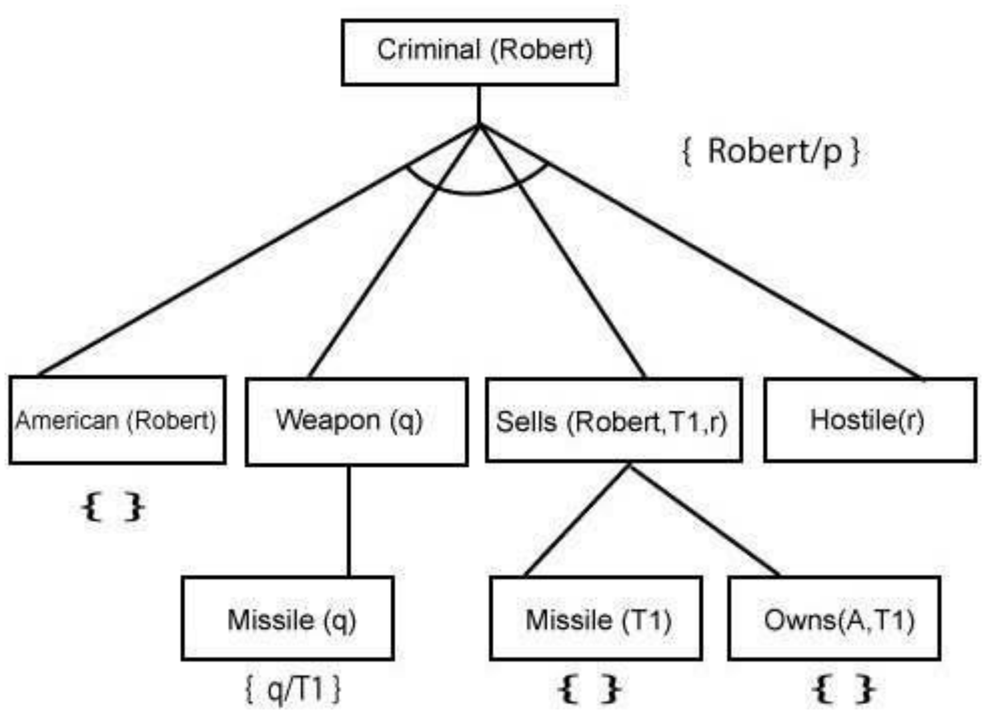


Step-3:t At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.



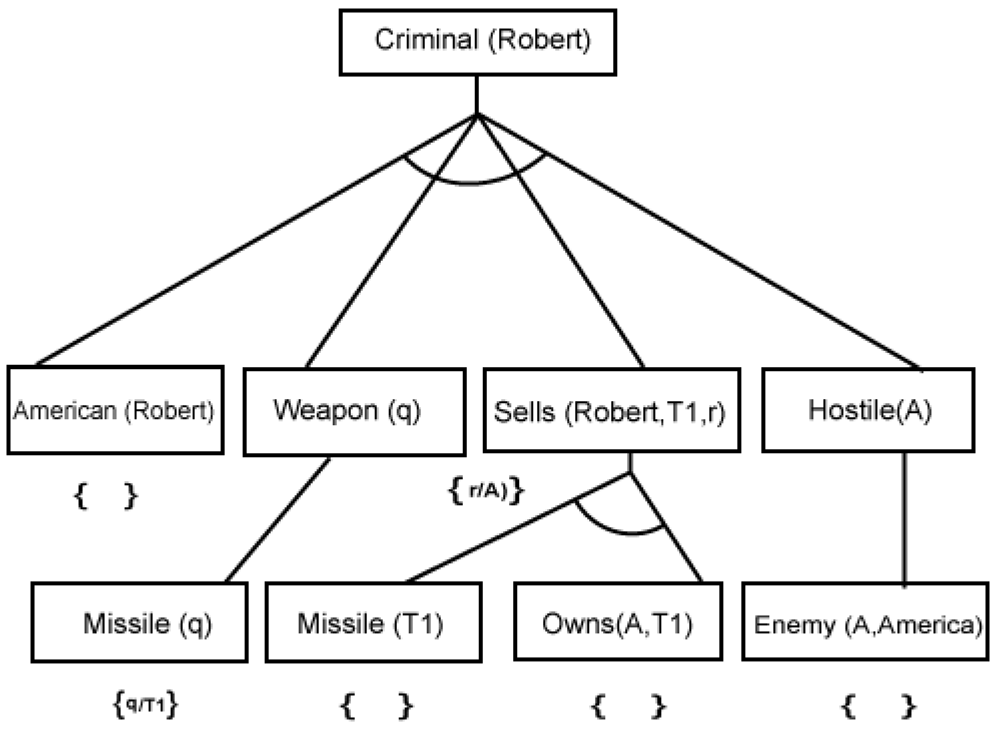
Step-4:

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.



Difference between backward chaining and forward chaining

Following is the difference between the forward chaining and backward chaining:

- Forward chaining as the name suggests, start from the known facts and move forward by applying inference rules to extract more data, and it continues until it reaches to the goal, whereas backward chaining starts from the goal, move backward by using inference rules to determine the facts that satisfy the goal.
- Forward chaining is called a data-driven inference technique, whereas backward chaining is called a goal-driven inference technique.
- Forward chaining is known as the down-up approach, whereas backward chaining is known as a top-down approach.

- Forward chaining uses breadth-first search strategy, whereas backward chaining uses depth-first search strategy.
- Forward and backward chaining both applies Modus ponens inference rule.
- Forward chaining can be used for tasks such as planning, design process monitoring, diagnosis, and classification, whereas backward chaining can be used for classification and diagnosis tasks.
- Forward chaining can be like an exhaustive search, whereas backward chaining tries to avoid the unnecessary path of reasoning.
- In forward-chaining there can be various ASK questions from the knowledge base, whereas in backward chaining there can be fewer ASK questions.
- Forward chaining is slow as it checks for all the rules, whereas backward chaining is fast as it checks few required rules only.

S. No.	Forward Chaining	Backward Chaining
1.	Forward chaining starts from known facts and applies inference rule to extract more data unit it reaches to the goal.	Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal.
2.	It is a bottom-up approach	It is a top-down approach
3.	Forward chaining is known as data-driven inference technique as we reach to the goal using the available data.	Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.
4.	Forward chaining reasoning applies a breadth-first search strategy.	Backward chaining reasoning applies a depth-first search strategy.
5.	Forward chaining tests for all the available rules	Backward chaining only tests for few required rules.

6.	Forward chaining is suitable for the planning, monitoring, control, and interpretation application.	Backward chaining is suitable for diagnostic, prescription, and debugging application.
7.	Forward chaining can generate an infinite number of possible conclusions.	Backward chaining generates a finite number of possible conclusions.
8.	It operates in the forward direction.	It operates in the backward direction.
9.	Forward chaining is aimed for any conclusion.	Backward chaining is only aimed for the required data.

ONTOLOGICAL ENGINEERING

In "toy" domains, the choice of representation is not that important; many choices will work. Complex domains such as shopping on the Internet or driving a car in traffic require more general and flexible representations. This chapter shows how to create these representations, concentrating on general concepts—such as *Events, Time, Physical Objects*, and *Beliefs*—that occur in many different domains. Representing these abstract concepts is sometimes called **ontological engineering**.

The prospect of representing *everything* in the world is daunting. Of course, we won't actually write a complete description of everything—that would be far too much for even a 1000-page textbook—but 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. This is analogous to the way that designers of an object-oriented programming framework (such as the Java Swing graphical framework) define general concepts like *Window*, expecting users to

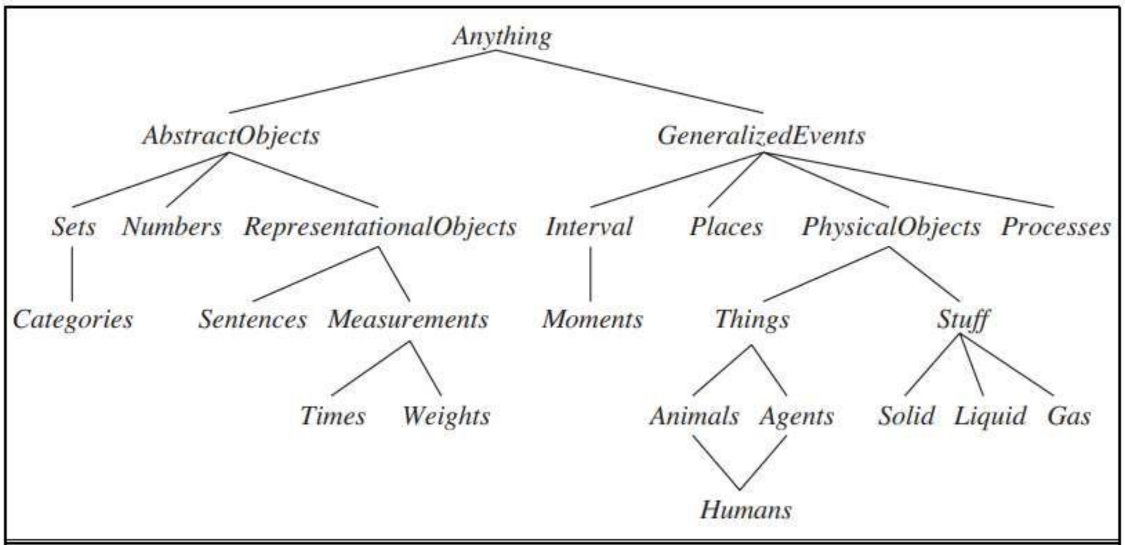


Figure 12.1 The upper ontology of the world, showing the topics to be covered later in

use these to define more specific concepts like *SpreadsheetWindow*. 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.

Before considering the ontology further, we should state one important caveat. We have elected to use first-order logic to discuss the content and organization of knowledge, although certain aspects of the real world are hard to capture in FOL. The principal difficulty is that most generalizations have exceptions or hold only to a degree. For example, although "tomatoes are red" is a useful rule, some tomatoes are green, yellow, or orange. Similar exceptions can be found to almost all the rules in this chapter. The ability to handle exceptions and uncertainty is extremely important, but is orthogonal to the task of understanding the general ontology. For this reason, we delay the discussion of exceptions until Section 12.5 of this chapter, and the more general topic of reasoning with uncertainty until Chapter 13.

Of what use is an upper ontology? Consider the ontology for circuits in Section 8.4.2. It makes many simplifying assumptions: time is omitted completely; signals are fixed and do not propagate; the structure of the circuit remains constant. A more general ontology would consider signals at particular times, and would include the wire lengths and propagation delays. This would allow us to simulate the timing properties of the circuit, and indeed such simulations are often carried out by circuit designers. We could also introduce more interesting classes of gates, for example, by describing the technology (TTL, CMOS, and so on) as well as the input–output specification. If we wanted to discuss reliability or diagnosis, we would include the possibility that the structure of the circuit or the properties of the gates might change spontaneously. To account for stray capacitances, we would need to represent where the wires are on the board.

If we look at the wumpus world, similar considerations apply. Although we do represent time, it has a simple structure: 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. We also used a *Pit* predicate to say which squares have pits. We could have allowed for different kinds of pits by having several individuals belonging to the class of pits, each having different properties. Similarly, we might want to allow for other animals besides wumpuses. It might not be possible to pin down the exact species from the available percepts, so we would need to build up a biological taxonomy to help the agent predict the behavior of cave-dwellers from scanty clues.

For any special-purpose ontology, it is possible to make changes like these to move toward greater generality. An obvious question then arises: do all these ontologies converge on a general-purpose ontology? After centuries of philosophical and computational investigation, the answer is "Maybe." In this section, we present one general-purpose ontology that synthesizes ideas from those centuries. 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 (with the addition of domain-specific axioms). 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 labor costs. The sentences describing time therefore must be capable of being combined with those describing spatial layout and must work equally well for nanoseconds and minutes and for angstroms and meters.

We should say up front that the enterprise of general ontological engineering has so far had only limited success. None of the top AI applications (as listed in Chapter 1) make use of a shared ontology—they all use special-purpose knowledge engineering. Social/political considerations can make it difficult for competing parties to agree on an ontology. As Tom Gruber (2004) says, "Every ontology is a treaty—a social agreement—among people with some common motive in sharing." When competing concerns outweigh the motivation for sharing, there can be no common ontology. Those ontologies that do exist have been created along four routes:

1. By a team of trained ontologist/logicians, who architect the ontology and write axioms. The CYC system was mostly built this way (Lenat and Guha, 1990).
2. By importing categories, attributes, and values from an existing database or databases. DBPEDIA was built by importing structured facts from Wikipedia (Bizer *et al.*, 2007).
3. By parsing text documents and extracting information from them. TEXTRUNNER was built by reading a large corpus of Web pages (Banko and Etzioni, 2008).
4. By enticing unskilled amateurs to enter commonsense knowledge. The OPENMIND system was built by volunteers who proposed facts in English (Singh *et al.*, 2002; Chklovski and Gil, 2005).

Categories and Objects

The organization of objects into categories

• 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

• Categories also serve to make predictions about objects once they are classified.

• Representation of categories in first-order logic:

• predicates eg. Basketball(b)

• Objects eg. Basketball

Categories serve to organize and simplify the knowledge base through inheritance

• Subclass relations organize categories into a taxonomy, or taxonomic hierarchy

• Subset(Basketballs, Balls)

Examples

• An object is a member of a category. BB9 Basketballs

• A category is a subclass of another category. Basketballs C 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) A Diameter(x) = 9.5" Axe Balls x Basketballs

• A category as a whole has some properties. Dogs DomesticatedSpecies

Objects

• The real world can be seen as consisting of primitive objects (e.g., atomic particles) and composite objects built from them.

- Stuff
- things

• some properties are intrinsic: they belong to the very substance of the object, rather than to the object as a whole

• Extrinsic properties-weight, length, shape, and so qn-are not retained under subdivision.

• A category of objects that includes in its definition only intrinsic properties is then a substance, or mass noun; a class that includes any extrinsic properties in its definition is a count noun.

Events

how situation calculus represents actions and their effects.

• Situation calculus-it was designed to describe a world in which actions are discrete, instantaneous, and happen one at a time.

• Event calculus-based on points of time rather than on situations.

• 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 describedasE1EFlyingsΛFlyer(E1,Shankar)ΛOrigin
(E1,SF)ΛDestination(E1,DC).

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

Processes

• Any process e that happens over an interval also happens over any subinterval

• Categories of events with this property are called process categories or liquid event categories.

(eЄProcesses)ΛHappens(e,(t1,t4))^(t1<t2<t3<t4) Happens(e,(t2,t3))

Mental Events and Mental Objects

Knowledge about one's own knowledge and reasoning processes is useful for controlling the inference

•The agent should know what are in its knowledge base and what are not

• What we need is a model of the mental objects that are in someone's head (or

some-thing's knowledge base) and of the mental processes that manipulate
those mental objects.
• propositional attitudes that an agent can have toward mental objects: attitudes such as Believes, Knows, Wants, Intends, and Informs.

Reasoning in Artificial intelligence

In previous topics, we have learned various ways of knowledge representation in artificial intelligence. Now we will learn the various ways to reason on this knowledge using different logical schemes.

Reasoning:

The reasoning is the mental process of deriving logical conclusion and making predictions from available knowledge, facts, and beliefs. Or we can say, "Reasoning is a way to infer facts from existing data." It is a general process of thinking rationally, to find valid conclusions.

In artificial intelligence, the reasoning is essential so that the machine can also think rationally as a human brain, and can perform like a human.

Types of Reasoning

In artificial intelligence, reasoning can be divided into the following categories:

- Deductive reasoning
- Inductive reasoning
- Abductive reasoning
- Common Sense Reasoning
- Monotonic Reasoning
- Non-monotonic Reasoning are the forms of propositional logic.

1. Deductive reasoning:

Deductive reasoning is deducing new information from logically related known information. It is the form of valid reasoning, which means the argument's conclusion must be true when the premises are true.

Deductive reasoning is a type of propositional logic in AI, and it requires various rules and facts. It is sometimes referred to as top-down reasoning, and contradictory to inductive reasoning.

In deductive reasoning, the truth of the premises guarantees the truth of the conclusion.

Deductive reasoning mostly starts from the general premises to the specific conclusion, which can be explained as below example.

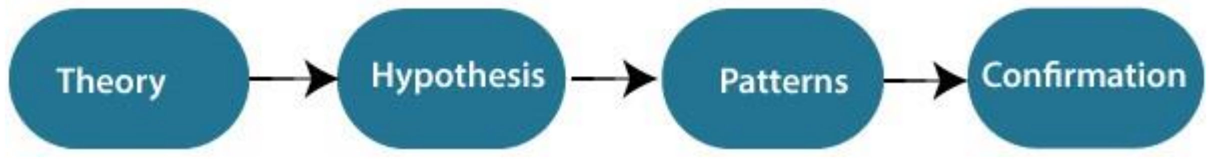
Example:

Premise-1: All the human eats veggies

Premise-2: Suresh is human.

Conclusion: Suresh eats veggies.

The general process of deductive reasoning is given below:



2. Inductive Reasoning:

Inductive reasoning is a form of reasoning to arrive at a conclusion using limited sets of facts by the process of generalization. It starts with the series of specific facts or data and reaches to a general statement or conclusion.

Inductive reasoning is a type of propositional logic, which is also known as cause-effect reasoning or bottom-up reasoning.

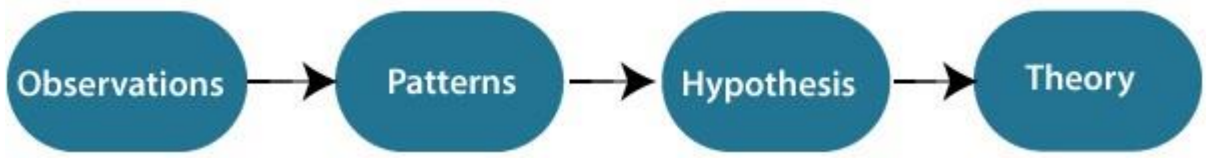
In inductive reasoning, we use historical data or various premises to generate a generic rule, for which premises support the conclusion.

In inductive reasoning, premises provide probable supports to the conclusion, so the truth of premises does not guarantee the truth of the conclusion.

Example:

Premise: All of the pigeons we have seen in the zoo are white.

Conclusion: Therefore, we can expect all the pigeons to be white.



3. Abductive reasoning:

Abductive reasoning is a form of logical reasoning which starts with single or multiple observations then seeks to find the most likely explanation or conclusion for the observation.

Abductive reasoning is an extension of deductive reasoning, but in abductive reasoning, the premises do not guarantee the conclusion.

Example:

Implication: Cricket ground is wet if it is raining

Axiom: Cricket ground is wet.

Conclusion It is raining.

4. Common Sense Reasoning

Common sense reasoning is an informal form of reasoning, which can be gained through experiences.

Common Sense reasoning simulates the human ability to make presumptions about events which occurs on every day.

It relies on good judgment rather than exact logic and operates on heuristic knowledge and heuristic rules.

Example:

1. One person can be at one place at a time.
2. If I put my hand in a fire, then it will burn.

The above two statements are examples of common sense reasoning which a human mind can easily understand and assume.

5. Monotonic Reasoning:

In monotonic reasoning, once the conclusion is taken, then it will remain the same even if we add some other information to existing information in our knowledge base. In monotonic reasoning, adding knowledge does not decrease the set of prepositions that can be derived.

To solve monotonic problems, we can derive the valid conclusion from the available facts only, and it will not be affected by new facts.

Monotonic reasoning is not useful for the real-time systems, as in real time, facts get changed, so we cannot use monotonic reasoning.

Monotonic reasoning is used in conventional reasoning systems, and a logic-based system is monotonic.

Any theorem proving is an example of monotonic reasoning.

Example:

- Earth revolves around the Sun.

It is a true fact, and it cannot be changed even if we add another sentence in knowledge base like, "The moon revolves around the earth" Or "Earth is not round," etc.

Advantages of Monotonic Reasoning:

- In monotonic reasoning, each old proof will always remain valid.
- If we deduce some facts from available facts, then it will remain valid for always.

Disadvantages of Monotonic Reasoning:

- We cannot represent the real world scenarios using Monotonic reasoning.
- Hypothesis knowledge cannot be expressed with monotonic reasoning, which means facts should be true.
- Since we can only derive conclusions from the old proofs, so new knowledge from the real world cannot be added.

6. Non-monotonic Reasoning

In Non-monotonic reasoning, some conclusions may be invalidated if we add some more information to our knowledge base.

Logic will be said as non-monotonic if some conclusions can be invalidated by adding more knowledge into our knowledge base.

Non-monotonic reasoning deals with incomplete and uncertain models.

"Human perceptions for various things in daily life, "is a general example of non-monotonic reasoning.

Example: Let suppose the knowledge base contains the following knowledge:

- Birds can fly
- Penguins cannot fly
- Pitty is a bird

So from the above sentences, we can conclude that Pitty can fly.

However, if we add one another sentence into knowledge base "Pitty is a penguin", which concludes "Pitty cannot fly", so it invalidates the above conclusion.

Advantages of Non-monotonic reasoning:

- For real-world systems such as Robot navigation, we can use non-monotonic reasoning.
- In Non-monotonic reasoning, we can choose probabilistic facts or can make assumptions.

Disadvantages of Non-monotonic Reasoning:

- In non-monotonic reasoning, the old facts may be invalidated by adding new sentences.
- It cannot be used for theorem proving.