1. Knowledge Representation

Knowledge is a collection of facts from some domain. Knowledge Representation and Reasoning (**KR, KRR**) represents information from the real world for a computer to understand and then utilize this knowledge to solve **complex real-life problems** like communicating with human beings in natural language. Knowledge representation in AI is not just about storing data in a database, it allows a machine to learn from that knowledge and behave intelligently like a human being.

Types of Knowledge:

There are 5 types of Knowledge such as:

1. **Declarative Knowledge**: It includes concepts, facts, and objects and is expressed in a declarative sentence.
2. **Structural Knowledge**: It is a basic problem-solving knowledge that describes the relationship between concepts and objects.
3. **Procedural Knowledge**: This is responsible for knowing how to do something and includes rules, strategies, procedures, etc.
4. **Meta Knowledge**: Meta Knowledge defines knowledge about other types of Knowledge.
5. **Heuristic Knowledge**: This represents some expert knowledge in the field or subject.

What to Represent:

Following are the kind of knowledge which needs to be represented in AI systems:

1. **Object:** All the facts about objects in our world domain.
2. **Events:** Events are the actions that occur in our world.
3. **Performance:** It describes behavior that involves knowledge about how to do things.

4. **Meta-knowledge:** It is knowledge about what we know.
5. **Facts:** Facts are the truths about the real world and what we represent.
6. **Knowledge-Base:** The central component of the knowledge-based agents is the knowledge base. It is represented as KB. A Knowledgebase is a group of Sentences (Here, sentences are used as a technical term and not identical with the English language).

Approaches to Knowledge Representation:

Properties:

A knowledge representation system should have the following properties:

1. **Representational Adequacy:** The ability to represent all kinds of knowledge that are needed in that domain.
2. **Inferential Adequacy:** The ability to manipulate the representational structures to derive new structures corresponding to new knowledge inferred from old.
3. **Inferential Efficiency:** The ability to incorporate additional information into the knowledge structure that can be used to focus the attention on the inference mechanisms in the most promising direction.
4. **Acquisitional Efficiency:** The ability to acquire new knowledge using automatic methods whenever possible rather than relying on human intervention.

Approaches :

1. **Simple relational knowledge:**

   It is the simplest way of storing facts which uses the relational method, and each fact about a set of the object is set out systematically in columns. This approach of knowledge representation is famous in database systems where the relationship between different entities is represented.

2. **Inheritable knowledge:**

In the inheritable knowledge approach, all data must be stored in a hierarchy of classes. All classes should be arranged in a generalized form or a hierarchal manner. In this approach, we apply inheritance property.

Elements inherit values from other members of a class. This approach contains inheritable knowledge which shows a relation between instance and class, and it is called instance relation. Every individual frame can represent the collection of attributes and their value.

In this approach, objects and values are represented in Boxed nodes. We use Arrows that point from objects to their values.

3. Inferential knowledge:

The inferential knowledge approach represents knowledge in the form of formal logic. This approach can be used to derive more facts. It guaranteed correctness.

4. Procedural knowledge:

The procedural knowledge approach uses small programs and codes which describe how to do specific things, and how to proceed. In this approach, one important rule is used which is the If-Then rule.

In this knowledge, we can use various coding languages such as LISP language and Prolog language. We can easily represent heuristic or domain-specific knowledge using this approach.

2. Representation and mappings :

Problem-solving requires a large amount of knowledge and some mechanism for manipulating that knowledge. Knowledge and representation play a central role in an intelligent system. Knowledge is a description of the world and the representation is the way knowledge is encoded.

Entities:

- **Facts:** These are the things we want to represent.
- **Representation of facts: These** are the things we will be able to manipulate.

Example :

Judy is a cat ……….fact

cat(Judy) ………representation of fact

Knowledge representation can be considered at two levels:

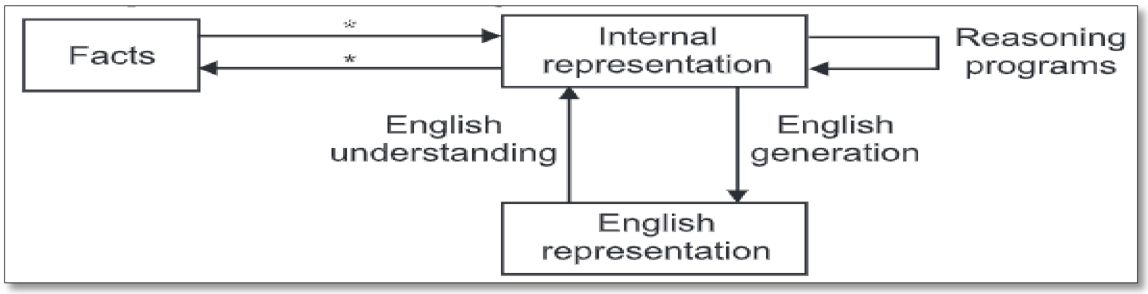
1. **Knowledge level** (at which facts are described).
2. **Symbol level** (at which the representation of the objects, defined in terms of symbols, can be manipulated in the programs).

Knowledge is a collection of facts from some domain. We need a representation of "facts" that can be manipulated by a program. Normal English is insufficient and too hard for the computer program to draw inferences in natural languages. Thus, some symbolic representation is necessary.

Therefore, we must be able to map "facts to symbols" and "symbols to facts" using forward and backward representation mapping.

Example :

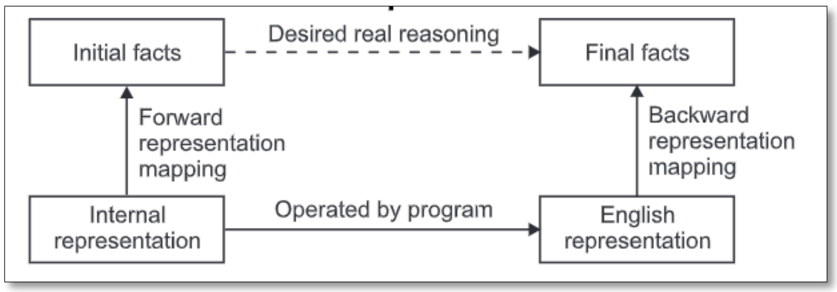
Consider an English sentence.



Facts	**Representation**
Judy is a cat	A fact represented in an English sentence.
cat (Judy)	Using forward mapping (maps facts to representation)
∀x:cat(x) → hastail(x)	A logical representation of the fact that “All cats have tails”.

Forward and backward representation:

The forward representation mapping maps from facts to representation. The backward representation mapping goes the other way, from representations to facts.



The dotted line on top indicates the abstract reasoning process that a program is intended to model.

The solid line on the bottom indicates the concrete reasoning process that the program performs.

Example :

Consider English sentence:

Judy is a cat

The fact represented by that English sentence can also be represented in logic as:

cat(Judy)

Suppose that we also have a logical representation of the fact that all cats have tails.

Every cat has a tail => ∀x:cat(x) → hastail(x)

Then, using the deductive mechanism of logic, we may generate the new representation object:

hastail(judy)

Using an appropriate backward mapping function, we could then generate the English sentence.

Judy has a tail ,is a new knowledge.

3. Issues in knowledge Representation:

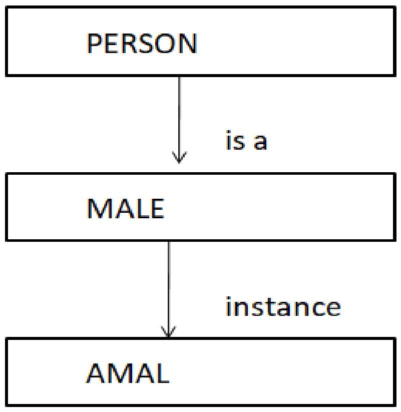
Many issues arise while using KR techniques. Some of them are:

1. **Important Attributes:** Any attributes of objects, it is so basic that they occur in almost every problem domain.
2. **Relationship among Attributes:** Any relationship that exists among attributes.
3. **Choosing Granularity:** At what level of details should the knowledge be represented?
4. **Set of Objects:** How sets of objects are represented?
5. **Finding Right Structure:** How can relevant parts be accessed, when a large amount of knowledge is given.

1. Important Attributes:

There is two attributes "instance" and "is a", that is of general importance. These attributes are important as they support property inheritance.

- instance=indicates class membership
- is a=indicates class inclusion.



2. Relationship among Attributes:

The attributes to describe objects are themselves entities they represent. The relationship between the attributes of an object, independent of specific knowledge they encode, may hold properties like:

a) **Inverses:** Entities in the world are related to each other in many different ways. This is about consistency check, while a value is added to one attribute.

b) **Existence in an `is a` hierarchy:** This is about generalization-specialization, like classes of objects and specialized subsets of those classes. These are attributes and specialization of attributes. These generalization-specialization relationships for attribute support inheritance.

c) **Techniques for reasoning about values:** This is about reasoning values of attributes not given explicitly.

Several kinds of information are used in reasoning, like

- **Age -** Of a person cannot be greater than the age of a person's parents.
- **Height -** It must be in a unit of length, cannot be in kg. These values are often specified when a knowledge base is created.

d) **Single valued attributes:** These are attributes that take unique values. In an employee database, an employee id is a unique value assigned to each employee.

3. Choosing Granularity:

At what level should the knowledge be represented and what are the primitives?

High-level facts may be adequate for inference while low-level primitives may require a lot of storage.

Example:

Suppose, we are interested in the fact.

John spotted Merry, which can be represented as

spotted (agent (John), object (Merry)).

Such a representation is easy to answer. **Who spotted Merry?**

Suppose we want to know.

Did John see Anu?

Given only one fact, we cannot discover that answer, we can add other facts.

spotted (X, Y) → saw (X, Y)

We can now infer the answer to the question.

4. **Representing a set of objects:**

Certain properties of objects are true as a member of a set but not as an individual.

Example: Page number written on the page of the book.

5. **Finding the right structure as needed:**

To describe a particular situation, it is always important to find access to the right structure. This can be done by selecting an initial structure and then revising the choice.

First-Order Logic in Artificial intelligence

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:
  - **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, ...
  - **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
  - **Function:** Father of, best ഫ്രണ്ട്, end of, ...
- As a natural language, first-order logic also has two main parts:
  - **c. Syntax**
  - **d. 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).

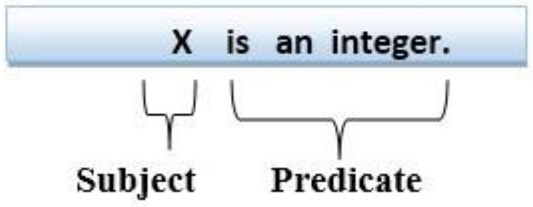
Complex Sentences:

- 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:
  e. **Universal Quantifier, (for all, everyone, everything)**
  f. **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 ∀, which resembles an inverted A.

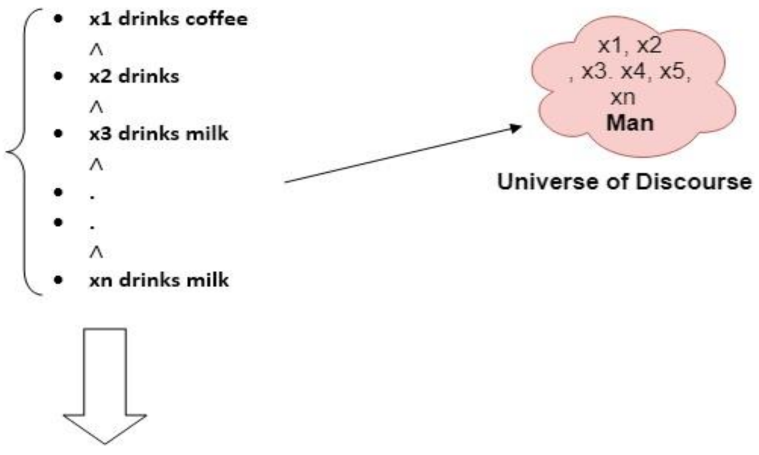
Note: In universal quantifier we use implication "→". If x is a variable, then ∀x is read as:

- **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:



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.

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.

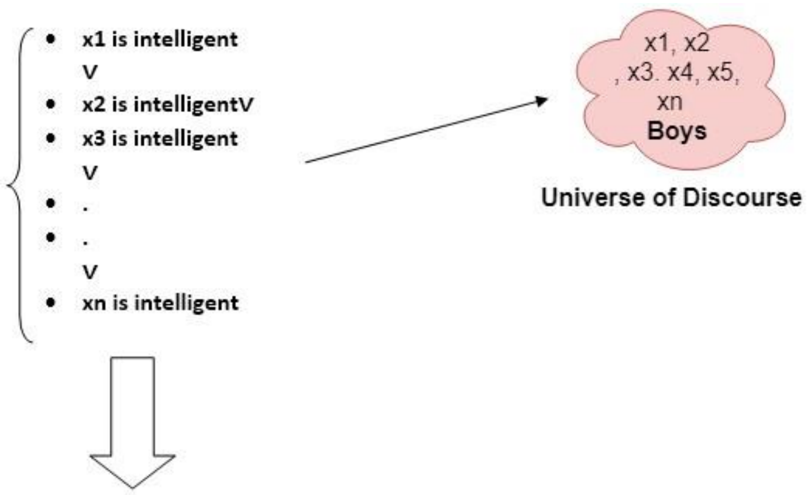
It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

Note: In Existential quantifier we always use AND or Conjunction symbol (∧).
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.



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

∃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. ∀**x bird(x) →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 ∀, and it will be represented as follows: **∀x man(x) → 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 ∃**, and it will be represented as**: **∃x boys(x) → 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 ∀ **with negation, so** following representation for this: **¬∀ (x) * student(x) → like(x, Mathematics) ∧ 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: **∃(x) * student(x) → failed (x, Mathematics) ∧∀ (y) [¬(x==y) ∧ student(y) → ¬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.

Propositional logic in Artificial intelligence

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Example:

1. a) It is Sunday.
2. b) The Sun rises from West (False proposition)
3. c) 3+3= 7(False proposition)
4. d) 5 is a prime number.

Following are some basic facts about propositional logic:

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and **logical connectives**.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
- A proposition formula which is always false is called **Contradiction**.

- Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

a. **Atomic Propositions**

b. **Compound propositions**

- **Atomic Proposition:** Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.

Example:

1. a) 2+2 is 4, it is an atomic proposition as it is a **true** fact.
2. b) "The Sun is cold" is also a proposition as it is a **false** fact.

- **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

1. a) "It is raining today, and street is wet."
2. b) "Ankit is a doctor, and his clinic is in Mumbai."

Logical Connectives:

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

1. **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
2. **Conjunction:** A sentence which has ∧ connective such as, **P ∧ Q** is called a conjunction.
   Example: Rohan is intelligent and hardworking. It can be written as,
   P= Rohan is intelligent,
   Q= Rohan is hardworking. → P∧ Q.
3. **Disjunction:** A sentence which has ∨ connective, such as **P ∨ Q**. is called disjunction, where P and Q are the propositions.
   Example: "Ritika is a doctor or Engineer",
   Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as **P ∨ Q.**
4. **Implication:** A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as
   If it is raining, then the street is wet.
   Let P= It is raining, and Q= Street is wet, so it is represented as P → Q
5. **Biconditional:** A sentence such as **P⇔ Q is a Biconditional sentence, example If I am breathing, then I am alive**
   P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

Following is the summarized table for Propositional Logic Connectives:

Connective symbols	Word	Technical term	Example
∧	AND	Conjunction	A ∧ B
∨	OR	Disjunction	A ∨ B
→	Implies	Implication	A → B
⇔	If and only if	Biconditional	A⇔ B
¬ or ~	Not	Negation	¬ A or ¬ B

Truth Table:

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is called **Truth table**. Following are the truth table for all logical connectives:

For Negation:

P	¬ P
True	**False**
False	**True**

For Conjunction:

P	Q	P∧ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

For disjunction:

P	Q	P V Q.
True	True	**True**
False	True	**True**
True	False	**True**
False	False	**False**

For Implication:

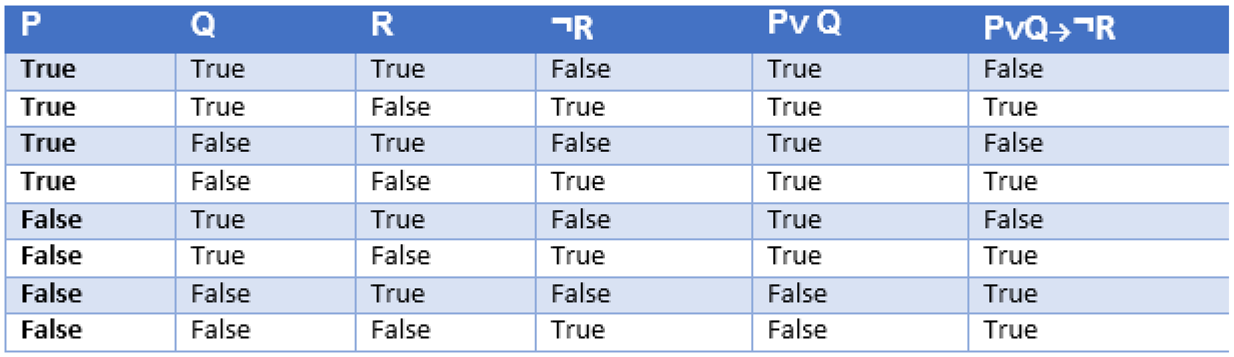
P	Q	P→ Q
True	True	**True**
True	False	**False**
False	True	**True**
False	False	**True**

For Biconditional:

P	Q	P⇔ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**True**

Truth table with three propositions:

We can build a proposition composing three propositions P, Q, and R. This truth table is made-up of 8n Tuples as we have taken three proposition symbols.



P	Q	R	¬R	P∨Q	P∨Q→¬R
True	True	True	False	True	False
True	True	False	True	True	True
True	False	True	False	True	False
True	False	False	True	True	True
False	True	True	False	True	False
False	True	False	True	True	True
False	False	True	False	False	True
False	False	False	True	False	True

Precedence of connectives:

Just like arithmetic operators, there is a precedence order for propositional connectors or logical operators. This order should be followed while evaluating a propositional problem. Following is the list of the precedence order for operators:

Precedence	**Operators**
First Precedence	Parenthesis
Second Precedence	Negation
Third Precedence	Conjunction(AND)
Fourth Precedence	Disjunction(OR)

Fifth Precedence	Implication
Six Precedence	Biconditional

Note: For better understanding use parenthesis to make sure of the correct interpretations. Such as ¬R∨Q, It can be interpreted as (¬R) ∨Q.

Logical equivalence:

Logical equivalence is one of the features of propositional logic. Two propositions are said to be logically equivalent if and only if the columns in the truth table are identical to each other.

Let's take two propositions A and B, so for logical equivalence, we can write it as A⇔B. In below truth table we can see that column for ¬A∨ B and A→B, are identical hence A is Equivalent to B

A	B	¬A	¬A∨ B	A→B
T	T	F	**T**	**T**
T	F	F	**F**	**F**
F	T	T	**T**	**T**
F	F	T	**T**	**T**

Properties of Operators:

- **Commutativity:**
  - P∧ Q= Q ∧ P, or
  - P ∨ Q = Q ∨ P.
- **Associativity:**
  - (P ∧ Q) ∧ R= P ∧ (Q ∧ R),
  - (P ∨ Q) ∨ R= P ∨ (Q ∨ R)
- **Identity element:**

- Distributive:
  - $P \land True = P$,
  - $P \lor True = True$.
- **Distributive:**
  - $P \land (Q \lor R) = (P \land Q) \lor (P \land R)$.
  - $P \lor (Q \land R) = (P \lor Q) \land (P \lor R)$.
- **DE Morgan's Law:**
  - $\neg (P \land Q) = (\neg P) \lor (\neg Q)$
  - $\neg (P \lor Q) = (\neg P) \land (\neg Q)$.
- **Double-negation elimination:**
  - $\neg (\neg P) = P$.

Limitations of Propositional logic:

- We cannot represent relations like ALL, some, or none with propositional logic. Example:

1. **All the girls are intelligent.**
2. **Some apples are sweet.**

- Propositional logic has limited expressive power.
- In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

2. Using predicate logic

1.Representing simple facts in logic:

In predicate logic, we can represent real-world facts as statements written as wff's.

Example :

Consider the following sentences:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

Convert given facts into predicate logic(First-order logic):

1. Marcus was a man.

   man(Marcus)

2. Marcus was a Pompeian.

   pompeian(Marcus)

3. All Pompeians 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.

   ∀x: Roman(x) → loyalto(x, Caesar) V hate(x, Caesar)
6. Everyone is loyal to someone.

   ∀x :∃y: Ioyalto(x,y)
7. People only try to assassinate rulers they are not loyal to.

   ∀x :∀ y : man(x) ∧ ruler(y) ∧ tryassassinate(x,y) → ¬ Ioyalto(x,y)
8. Marcus tried to assassinate Caesar.

   tryassassinate (Marcus, Caesar)

Question:

Was marcus loyal to caesar?

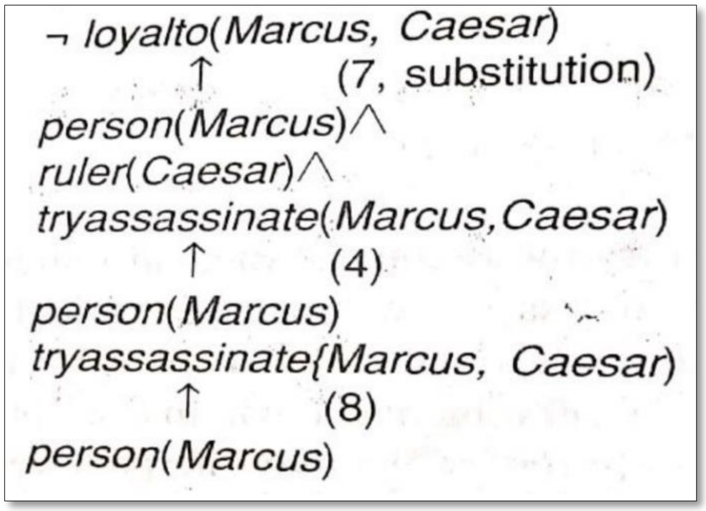
Rules:

1. man(Marcus)
2. Pompeian(Marcus)
3. ∀x: Pompeian(x) → Roman(x)
4. ruler(Caesar)
5. ∀x: Roman(x) → loyalto(X. Caesar) V hate(x, Caesar)
6. ∀x :∃y: Ioyalto(x,y)
7. ∀x :∀ y : man(x) ∧ ruler(y) ∧ tryassassinate(x,y) → ¬ Ioyalto(x,y)
8. tryassassinate (Marcus, Caesar)

Answer :

¬ loyalto(Marcus,Caesar)

Proof:



¬ loyalto(Marcus, Caesar)
↑ (7, substitution)
person(Marcus)∧
ruler(Caesar)∧
tryassassinate(Marcus,Caesar)
↑ (4)
person(Marcus)
tryassassinate{Marcus, Caesar)
↑ (8)
person(Marcus)

2.Representing instances and IS A relationship:

Specific attributes **instance** and **is a** play important role particularly in a useful form of reasoning called property inheritance. The predicates instance and is explicitly captured the relationship they are used to express, namely class membership and class inclusion.

Example:

1. man(Marcus)
2. Pompeian(Marcus)
3. ∀x: Pompeian(x) → Roman(x)
4. ruler(Caesar)

Part 1(Representation):

1. Man(Marcus).
2. Pompeian(Marcus).
3. ∀x: Pompeian(x) → Roman(x).
4. ruler(Caesar).

In these representations, class membership is represented with unary predicates, each of which corresponds to a class.

Part 2(Representation):

1. instance(Marcus, Man).
2. instance(Marcus ,Pompeian).
3. ∀x: instance(x ,Pompeian)→ instance(x,Roman).
4. instance(Caesar,ruler).

The second part contains representations that use the **instance** predicate explicitly. The predicate **instance** is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit **is a** predicate.

Part 3(Representation):

1. instance(Marcus, Man).
2. instance(Marcus ,Pompeian).
3. is a(Pompeian,Roman).
4. instance(Caesar,ruler).
5. ∀x: ∀y: ∀z: instance(x ,y) ^ is a(y,z)→ instance(x,z).

The third part contains representations that use both the instance and is a predicate explicitly. The use of the is a predicate simplifies the representation of sentence 3, but it requires that one additional axiom be provided.

3.<u>Computable functions and predicates:</u>

To express simple facts, such as the following greater-than and less-than relationships:

gut(1,0)	**lt(0,1)**
gt(2,1)	**lt(1,2)**

…………………………

It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of **gt(2+3,1)**.To do so requires that we first compute the value of the plus functions given the arguments 2 and 3, and then send the arguments 5 and 1 to gt.

Example:

Consider the following sentences:

1. Marcus was a man
2. Marcus was a Pompeian
3. Marcus was born in 40 A.D.
4. All men are mortal
5. All pompeians died when the volcano erupted in 79 A.D.
6. No mortal lives longer than 150 years.
7. It is now 1991.
8. Alive means not dead
9. If someone dies, then he is dead at all later times.

Convert given facts into predicate logic(First-order logic):

1. Marcus was a man

   man(Marcus)
2. Marcus was a Pompeian

   Pompeian(Marcus)
3. Marcus was born in 40 A.D.

   Born(Marcus,40)
4. All men are mortal.

   ∀x:man(x) → mortal(x)
5. All pompeians died when the volcano erupted in 79 A.D.

Erupted(volcano,79) ^ ∀x:[Pompeian(x) → died(x,79)]

6. No mortal lives longer than 150 years.

   ∀x:∀t1:∀t2:mortal(x) ^ born(x,t1) ^ gt(t2-t1,150) →dead(x,t2)

7. It is now 1991.

   Now=1991

8. Alive means not dead

   ∀x:∀t:[alive(x,t) → ~dead(x,t)] ^ [~dead(x,t) →alive(x,t)]

9. If someone dies, then he is dead at all later times.

   ∀x:∀t1:∀t2:died(x,t1) ^ gt(t2,t1) →dead(x,t2)]

Question :

Is Marcus alive?

Rules:

1. man(marcus)
2. pompeian(marcus)
3. born(marcus,40)
4. ∀x:man(x) → mortal(x)
5. ∀x:[pompeian(x) → died(x,79)]
6. erupted(volcano,79)
7. ∀x:∀t1:∀t2:mortal(x) ^ born(x,t1) ^ gt(t2-t1,150) →dead(x,t2)
8. now=1991
9. ∀x:∀t:[alive(x,t) → ~dead(x,t)] ^ [~dead(x,t) →alive(x,t)]

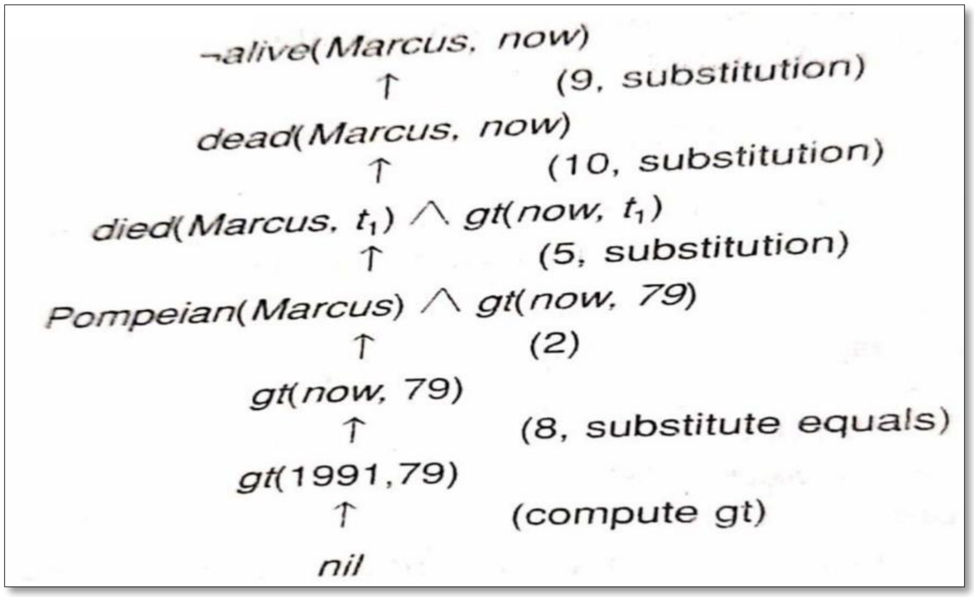
10. ∀x:∀t1:∀t2:died(x,t1) ^ gt(t2,t1) →dead(x,t2)]

Answer :

Marcus is dead

~alive(Marcus,now)

Proof:



$\neg alive(Marcus, now)$

$\uparrow$ (9, substitution)

$dead(Marcus, now)$

$\uparrow$ (10, substitution)

$died(Marcus, t_1) \wedge gt(now, t_1)$

$\uparrow$ (5, substitution)

$Pompeian(Marcus) \wedge gt(now, 79)$

$\uparrow$ (2)

$gt(now, 79)$

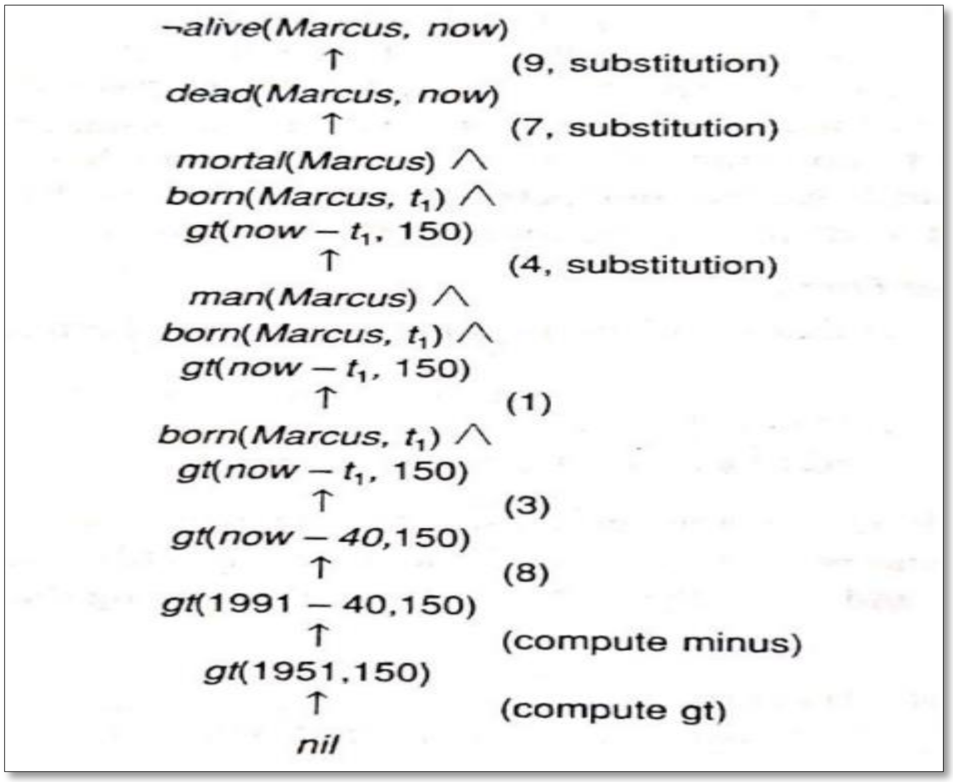
$\uparrow$ (8, substitute equals)

$gt(1991,79)$

$\uparrow$ (compute gt)

nil

Or



$$\begin{tikzcd}
\neg alive(Marcus, now) \\
dead(Marcus, now) \arrow[u, "\text{(9, substitution)}"'] \\
mortal(Marcus) \wedge born(Marcus, t_1) \wedge gt(now - t_1, 150) \arrow[u, "\text{(7, substitution)}"'] \\
man(Marcus) \wedge born(Marcus, t_1) \wedge gt(now - t_1, 150) \arrow[u, "\text{(4, substitution)}"'] \\
born(Marcus, t_1) \wedge gt(now - t_1, 150) \arrow[u, "(1)"'] \\
gt(now - 40, 150) \arrow[u, "(3)"'] \\
gt(1991 - 40, 150) \arrow[u, "(8)"'] \\
gt(1951, 150) \arrow[u, "\text{(compute minus)}"'] \\
nil \arrow[u, "\text{(compute gt)}"']
\end{tikzcd}$$

4.Resolution:

Resolution is a theorem proving technique that proofs by contradictions. It is used, if there are various statements are given and need to prove a conclusion of these statements. Unification is a key concept in proofs by resolution. It is a single inference rule which can efficiently operate on conjunctive normal form or clause form.

Steps:

1. Negate the statement to be proved.
2. Convert given facts into predicate logic(First-order logic).
3. Convert predicate logic into CNF(Conjunctive normal form).

4. Draw a resolution graph.

Convert predicate logic into CNF(Conjunctive normal form):

Rules:

1. Eliminate →, using the fact that a → b is equivalent to

   ¬ a V b.

2. Eliminate ↔, using the fact that a ↔ b is equivalent to

   (a → b) ∧ (b → a).

3. Reduce the scope of each ¬ to a single term:

   a) ¬ (¬ p) = p

   b) ¬ (a ∧b) = ¬ a V ¬ b

   c) ¬ (a V b) = ¬ a ∧ ¬ b

   d) ¬ ∀x: P(x) = ∃x: ¬ P(x)

   e) ¬∃x: P(x) = ∀x: ¬P(x)

4. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff.
5. Replace the Existential quantifier with the Skolem constant.
   Example: ∃xRich(x) =Rich(G1)
6. Drop universal quantifier.

Example:

Consider the following sentences:

1. Marcus was a man.

2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

Prove that Marcus hates Caesar using resolution?

Solution:

Step 1: Negate the statement to be proved

Marcus hate Caesar

↓ **convert statement into predicate logic**

hate(Marcus,Caesar)

↓ **negate the predicate logic statement**

¬ hate(Marcus,Caesar)

<u>Step 2 : Convert given facts into predicate logic(First order logic)</u>

The predicate logic statements are:

1. man(Marcus)
2. Pompeian(Marcus)
3. ∀x: Pompeian(x) → Roman(x)
4. ruler(Caesar)
5. ∀x: Roman(x) → loyalto(X. Caesar) V hate(x, Caesar)
6. ∀x :∃y: Ioyalto(x,y)
7. ∀x :∀ y : man(x) ∧ ruler(y) ∧ tryassassinate(x,y) → ¬ Ioyalto(x,y)
8. tryassassinate (Marcus, Caesar)

<u>Step 3:Convert predicate logic into CNF(Conjunctive normal form).</u>

1. man(Marcus)

   man(Marcus)

2. Pompeian(Marcus)

   Pompeian(Marcus)

3. ∀x: Pompeian(x) → Roman(x)

   Apply rule 1 and 6

   ¬ Pompeian(x) V Roman(x)

4. ruler(Caesar)

   ruler(Caesar)

5. ∀x: Roman(x) → loyalto(x,Caesar) V hate(x, Caesar)

   Apply rule 1 and 6

   ¬ Roman(x) V loyalto(x. Caesar) V hate(x, Caesar)

6. ∀x :∃y: Ioyalto(x,y)

   Apply rule 5 and 6

   Ioyalto(x,fl(x))

7. ∀x :∀ y : man(x) ∧ ruler(y) ∧ tryassassinate(x,y) → ¬ Ioyalto(x,y)

   Apply Rule 1 and 6

   ¬ [man(x) ∧ ruler(y) ∧ tryassassinate(x,y)]V ¬ Ioyalto(x,y)

   Apply rule 3b

   ¬ man(x) V ¬ ruler(y) V ¬ tryassassinate(x,y) V ¬ Ioyalto(x,y)

8. tryassassinate (Marcus, Caesar)

   tryassassinate (Marcus, Caesar)

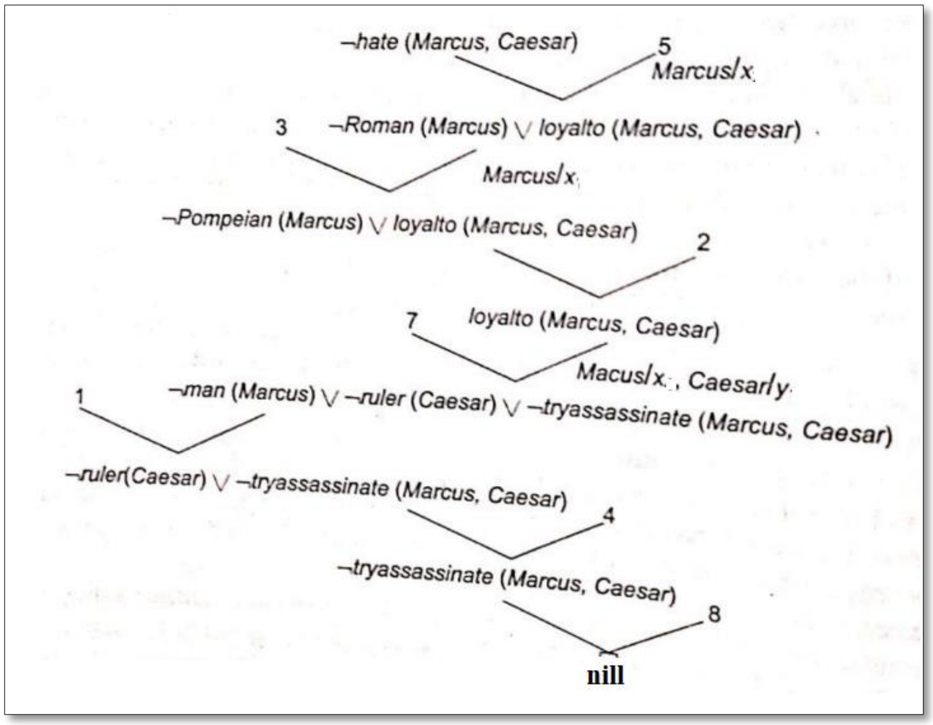
CNF forms:

1. man(Marcus)
2. Pompeian(Marcus)
3. ¬ Pompeian(x) V Roman(x)
4. ruler(Caesar)
5. ¬ Roman(x) V loyalto(x. Caesar) V hate(x, Caesar)
6. Ioyalto(x,fl(x))

7. ¬ man(x) V ¬ ruler(y) V ¬ tryassassinate(x,y) V ¬ Ioyalto(x,y)

8. tryassassinate (Marcus, Caesar)

Step 4: Draw resolution graph :


¬hate (Marcus, Caesar)
5
Marcus/x
3
¬Roman (Marcus) ∨ loyalto (Marcus, Caesar)
Marcus/x
¬Pompeian (Marcus) ∨ loyalto (Marcus, Caesar)
2
7
loyalto (Marcus, Caesar)
Macus/x , Caesar/y
1
¬man (Marcus) ∨ ¬ruler (Caesar) ∨ ¬tryassassinate (Marcus, Caesar)
¬ruler(Caesar) ∨ ¬tryassassinate (Marcus, Caesar)
4
¬tryassassinate (Marcus, Caesar)
8
nill

The Unification Algorithm:

Unification means making expressions look identical. It can be done with the process of substitution.

Conditions for Unification:

1. Predicate symbol must be the same, atoms or expressions with different predicate symbol can never be unified.
2. The number of Arguments in both expressions must be identical.

3. Unification will fail if there are two similar variables present in the same expression.

Example:

Let's say there are two different expressions,

P(x, y) ……………..1

P(a, f(z)) ……………2

Here 1 and 2 are identical if x is replaced with a, and y is replaced with f(z), 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]**.

Example 2:

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

Here, L1 = knows(Richard, x), and L2 = knows(Richard, John)
S_0 =>{knows(Richard,x);knows(Richard,John)}
SUBST={John/x}
S_1 => { knows(Richard, John); knows(Richard, John)}, Successfully unified.
Unifier: {John/x}.

Algorithm:

```
Unify (L1, L2)
1. If L1 or L2 are both variables or constants, then:
    (a) If L1 and L2 are identical, then return NIL.
    (b) Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return (L2/L1).
    (c) Else if L2 is a variable, then if L2 occurs in L1 then return {FAIL}, else return (L1/L2).
    (d) Else return {FAIL}.
2. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL}.
3. If LI and L2 have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify
L1 and L2.)
5. For i ← 1 to number of arguments in L1 :
    (a) Call Unify with the ith argument of L1 and the ith argument of L2, putting result in S.
    (b) If S contains FAIL then return {FAIL}.
    (c) If S is not equal to NIL then:
        (i) Apply S to the remainder of both L1 and L2.
        (ii) SUBST: = APPEND(S, SUBST).
6. Return SUBST.
```

Determining that two literals are contradictory- they are if one of them can be unified with the negation of the other. So, for example, man(x) and ¬ man(Marcus) are contradictory, since man(x) and man(Marcus) can be unified.

To use resolution for expressions in the predicate logic, use the unification algorithm to locate pairs of literals that cancel out. Need to use the unifier produced by the unification algorithm to generate the resolvent clause.

Example:

Suppose to resolve two clauses:

1. man(Marcus)
2. ¬ man(x1) V mortal(x1)

The literal man(Marcus) can be unified with the literal **man(x1)** with the substitution **Marcus/x1**, telling us that for x1 = Marcus, ¬ man(Marcus) is false.

But cannot simply cancel out the two-man literals as we did in propositional logic and generate the resolvent **mortal(x1)**. Clause 2 says that for a given x1, either ¬ **man(x1) or mortal(x1).** So for it to be true, can now conclude only that **mortal(Marcus)** must be true.

5. Natural deduction:

Testing whether a proposition is a tautology by testing every possible truth assignment is expensive. We need a deductive system, which will allow us to construct proofs of tautologies in a step-by-step fashion. The system we will use is known as natural deduction. The system consists of a set of rules of inference for deriving consequences from premises.

Most rules come in one of two flavors: **introduction** or **elimination** rules. Introduction rules introduce the use of a logical operator, and elimination rules eliminate it.

Rules for Conjunction :

A ∧ B should be true if both A and B are true. Thus we have the following introduction rule.

$$\frac{A\ \mathit{true} \qquad B\ \mathit{true}}{A \wedge B\ \mathit{true}}\wedge\mathrm{I}$$

If we consider this as a complete definition, we should be able to recover both A and B if we know A ∧ B. We are thus led to two elimination rules.

$$\frac{A \wedge B\ \mathit{true}}{A\ \mathit{true}}\wedge\mathrm{E_L} \qquad \frac{A \wedge B\ \mathit{true}}{B\ \mathit{true}}\wedge\mathrm{E_R}$$

Formulated as a transformation or reduction between derivations we have

$$\dfrac{\dfrac{\overset{\mathcal{D}}{A\ \mathit{true}} \qquad \overset{\mathcal{E}}{B\ \mathit{true}}}{A \wedge B\ \mathit{true}}\wedge\mathrm{I}}{A\ \mathit{true}}\wedge\mathrm{E_L} \quad \Longrightarrow_R \quad \overset{\mathcal{D}}{A\ \mathit{true}}$$

and symmetrically

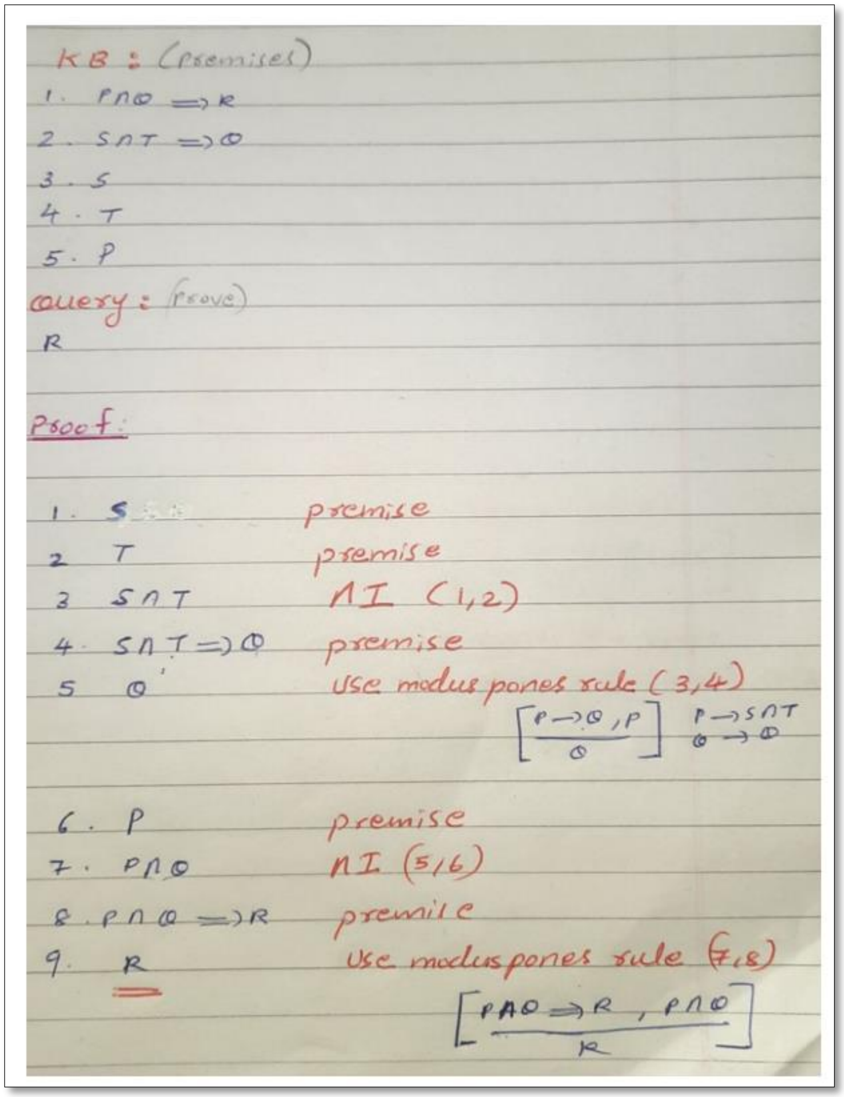
$$\dfrac{\dfrac{\overset{\mathcal{D}}{A\ \mathit{true}} \qquad \overset{\mathcal{E}}{B\ \mathit{true}}}{A \wedge B\ \mathit{true}}\wedge\mathrm{I}}{B\ \mathit{true}}\wedge\mathrm{E_R} \quad \Longrightarrow_R \quad \overset{\mathcal{E}}{B\ \mathit{true}}$$

Since local reductions are possible for both elimination rules for conjunction, our rules are locally sound.

To show that the rules are locally complete we show how to reintroduce conjunction from its components in the form of a local expansion.

$$\overset{\mathcal{D}}{A \wedge B\ \mathit{true}} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\overset{\mathcal{D}}{A \wedge B\ \mathit{true}}}{A\ \mathit{true}}\wedge\mathrm{E_L} \qquad \dfrac{\overset{\mathcal{D}}{A \wedge B\ \mathit{true}}}{B\ \mathit{true}}\wedge\mathrm{E_R}}{A \wedge B\ \mathit{true}}\wedge\mathrm{I}$$

Example:



KB : (premises)

1. $P \wedge Q \Rightarrow R$
2. $S \wedge T \Rightarrow Q$
3. S
4. T
5. P

Query : (prove)

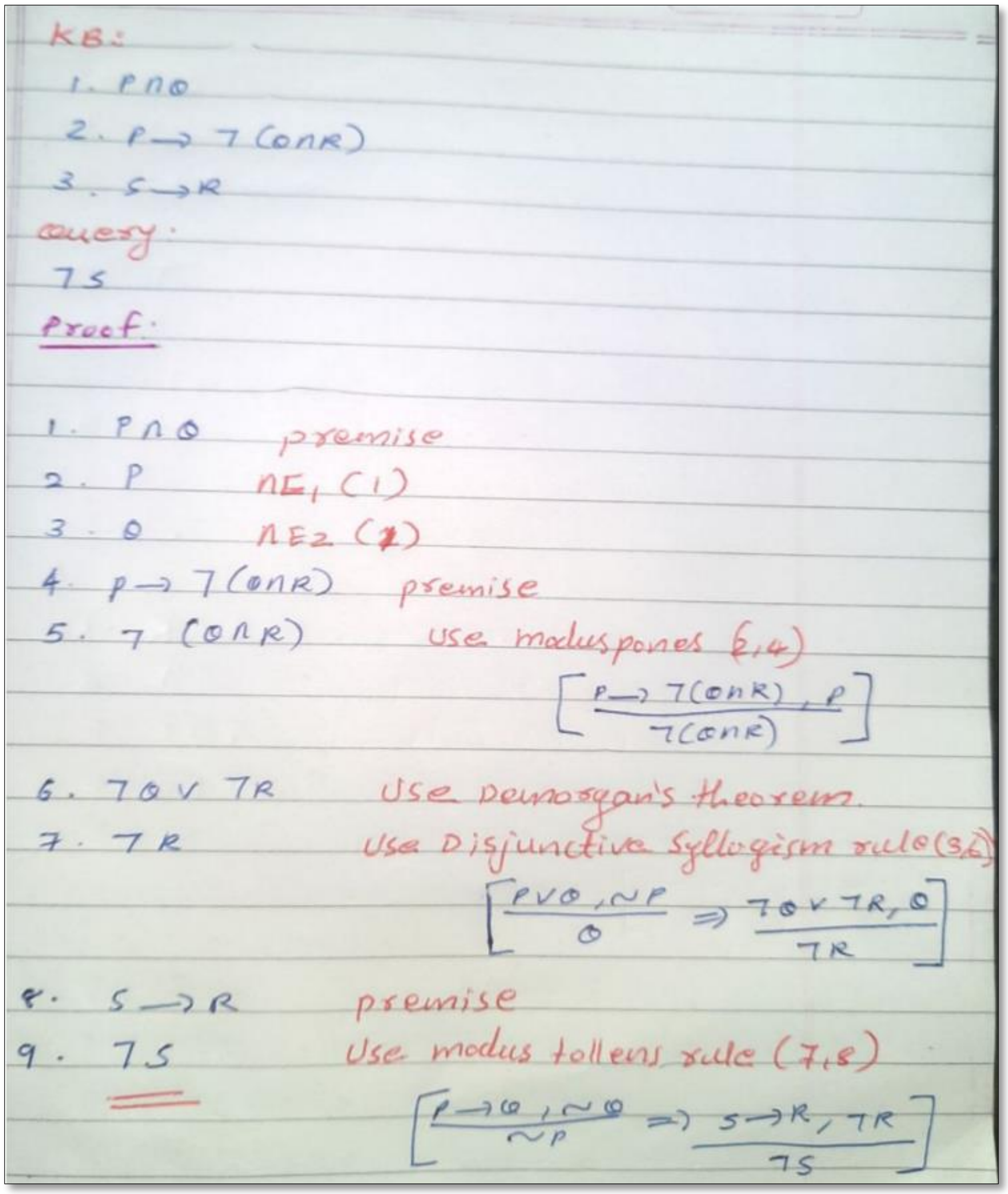
R

Proof:

1.	S	premise
2	T	premise
3	$S \wedge T$	$\wedge I$ (1,2)
4.	$S \wedge T \Rightarrow Q$	premise
5	Q	Use modus pones rule (3,4)
6.	P	premise
7.	$P \wedge Q$	$\wedge I$ (5,6)
8.	$P \wedge Q \Rightarrow R$	premile
9.	R	Use moduspones rule (7,8)

$$\left[\frac{P \rightarrow Q ,\ P}{Q}\right] \quad \begin{matrix} P \rightarrow S \wedge T \\ Q \rightarrow Q \end{matrix}$$

$$\left[\frac{P \wedge Q \Rightarrow R ,\ P \wedge Q}{R}\right]$$



KB:

1. $P \wedge Q$
2. $P \rightarrow \neg(Q \wedge R)$
3. $S \rightarrow R$

Query:

$\neg S$

Proof:

1. $P \wedge Q$ premise
2. P $\wedge E_1$ (1)
3. Q $\wedge E_2$ (1)
4. $P \rightarrow \neg(Q \wedge R)$ premise
5. $\neg(Q \wedge R)$ use modus pones (2,4)

$$\left[\frac{P \rightarrow \neg(Q \wedge R),\ P}{\neg(Q \wedge R)}\right]$$

6. $\neg Q \vee \neg R$ Use Demorgan's theorem.
7. $\neg R$ Use Disjunctive Syllogism rule (3,6)

$$\left[\frac{P \vee Q,\ \sim P}{Q} \Rightarrow \frac{\neg Q \vee \neg R,\ Q}{\neg R}\right]$$

8. $S \rightarrow R$ premise
9. $\neg S$ Use modus tollens rule (7,8)

$$\left[\frac{P \rightarrow Q,\ \sim Q}{\sim P} \Rightarrow \frac{S \rightarrow R,\ \neg R}{\neg S}\right]$$

3. Knowledge representation using rules

1. Logic programming :

It is a programming language paradigm in which logical assertions are viewed as programs. There are several logic programming systems in use today, the most popular of which is PROLOG.

A PROLOG program is described as a series of logical assertions, each of which is a Horn clause. In prolog, we compose the program using facts and rules.

Horn Clause :

A Horn clause is a clause that has at most one positive literal. Thus p, ~ p V q, and p → q are all horn clauses. Horn clause consists of the head (left-hand side) and body (right-hand side). The Head can have 0 or 1 predicate and the body can have a list of predicates. That means LHS has only a single literal and RHS can have more than one literals.

Syntax:

Start the statement (relationship or Object) with lowercase letters and end with '.' period (full stop) - for all facts, rules, and queries.

Written as :

h:- b (where b is p1, p2, p3, .., pn and ' :- ' operator is read as 'if')

Read as :

h is true *if* b is true. In other words, h is true if all the predicates on the right side are true.

Facts :

Facts are those statements that state the objects or describe the relationship between objects. Facts are also known as "unconditional horn clauses" and they are always true.

For instance, when we say **Marcus tried to assassinate Caesar**, we are showing the ' **tried to assassinate** ' relationship between two objects '**Marcus and 'Caesar',** and in prolog, this fact can be written as **tryassassinate (Marcus, Caesar).**

Rules :

Rules are the conditional statements about objects and their relationships.

Examples:

- All Pompeians were Romans.

  ∀x: Pompeian(x) → Roman(x) (in logic)

  roman(X) : - pompeian(X) (in prolog)

- All Romans were either loyal to Caesar or hated him.

∀x: Roman(x) → loyalto(x, Caesar) V hate(x, Caesar) (in logic)

$$\left.\begin{array}{l}\textbf{loyalto(X, Caesar) : - roman(X).}\\ \textbf{hate(X, Caesar) :- roman(X).}\end{array}\right\}\text{(in prolog)}$$

Queries/Goals :

Prolog queries are the questions asked by the user to prolog interpreters about facts and rules stored in its database. For the particular program, upon asking the query, the Prolog interpreter trace through the facts and rules it has been given in a top-down manner and tries to find a match for a given query. If it finds the matches it will report yes/true and if it doesn't then it will report no/false.

<u>Variables :</u>

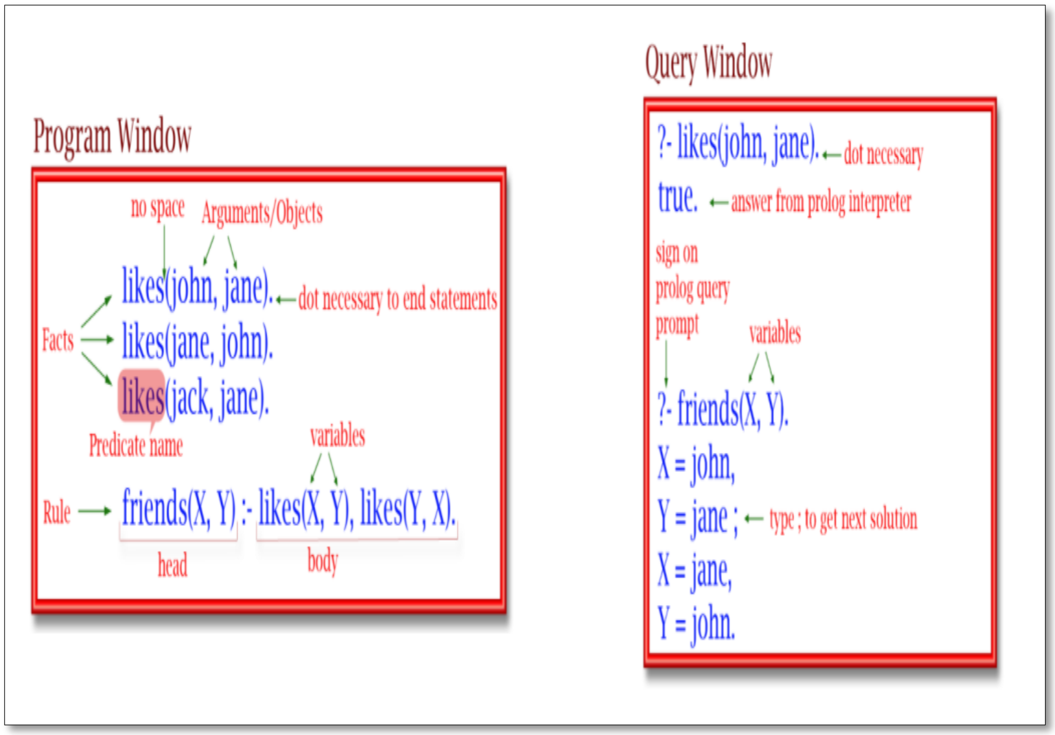
In prolog variables always start with an uppercase letter or an underscore and it can be written in head or body or the goals.

Example:

Who, What, X, Y, ABC, Abc, A12

<u>Structure:</u>

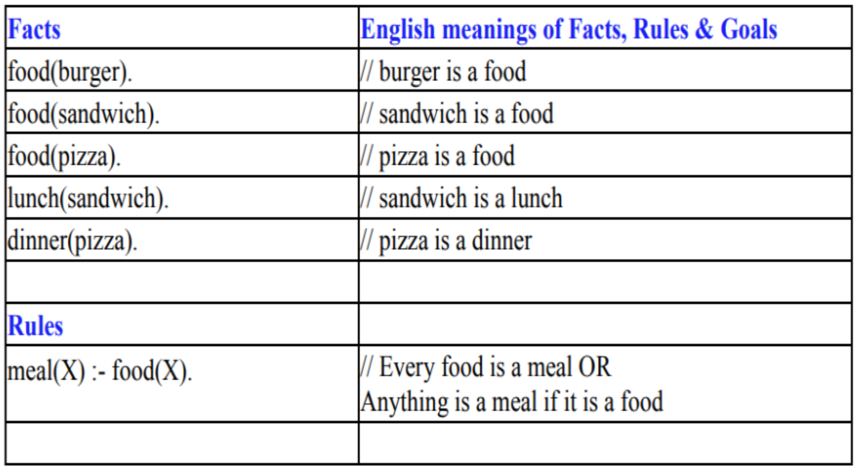
Prolog always performs depth-first-search, Matches facts & rules (i.e. knowledge base) in a top-down manner, and resolves the goals or subgoals in a left-to-right manner. The most important thing to keep in mind while writing a prolog program - "order of writing facts & rules always matters".


Program Window
no space
Arguments/Objects
dot necessary to end statements
Facts
Predicate name
variables
Rule
head
body
Query Window
dot necessary
answer from prolog interpreter
sign on prolog query prompt
variables
type ; to get next solution

```
likes(john, jane).
likes(jane, john).
likes(jack, jane).
friends(X, Y) :- likes(X, Y), likes(Y, X).
```

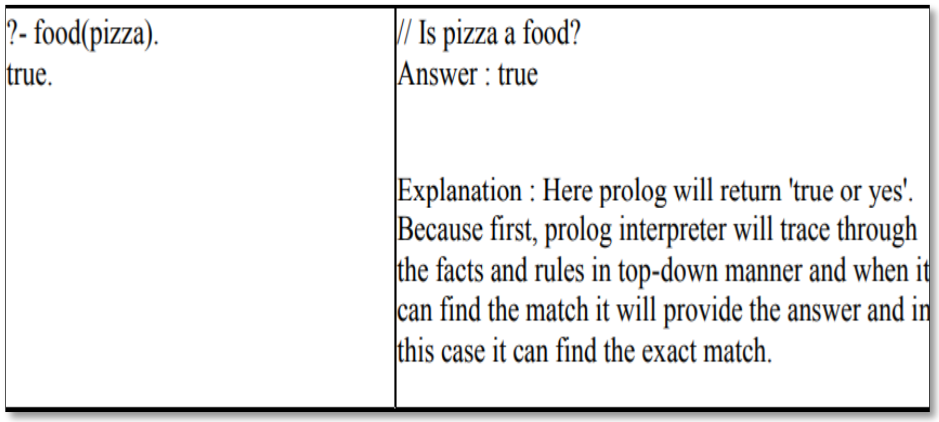
```
?- likes(john, jane).
true.
?- friends(X, Y).
X = john,
Y = jane ;
X = jane,
Y = john.
```

Example : Food

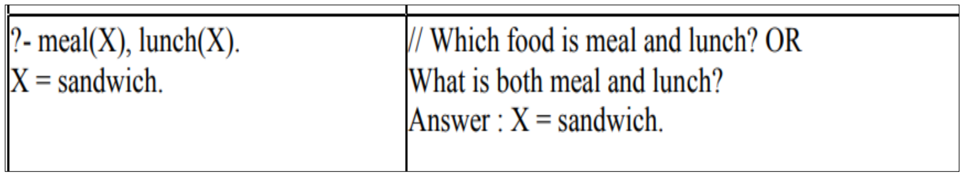


Facts	English meanings of Facts, Rules & Goals
food(burger).	// burger is a food
food(sandwich).	// sandwich is a food
food(pizza).	// pizza is a food
lunch(sandwich).	// sandwich is a lunch
dinner(pizza).	// pizza is a dinner
Rules	
meal(X) :- food(X).	// Every food is a meal OR Anything is a meal if it is a food

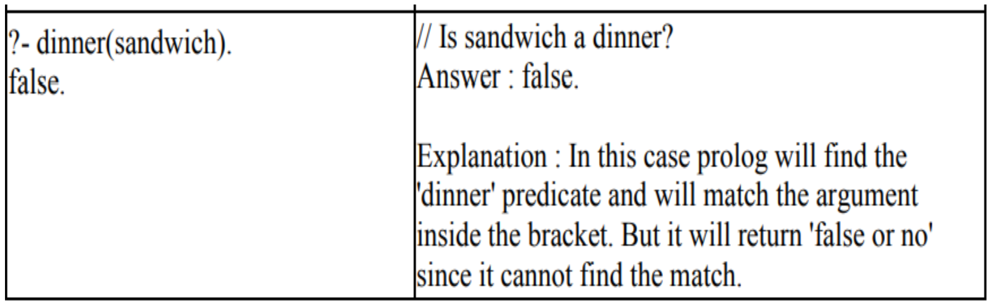
Queries / Goals & answers



?- food(pizza). true.	// Is pizza a food? Answer : true Explanation : Here prolog will return 'true or yes'. Because first, prolog interpreter will trace through the facts and rules in top-down manner and when it can find the match it will provide the answer and in this case it can find the exact match.



?- meal(X), lunch(X). X = sandwich.	// Which food is meal and lunch? OR What is both meal and lunch? Answer : X = sandwich.



?- dinner(sandwich). false.	// Is sandwich a dinner? Answer : false. Explanation : In this case prolog will find the 'dinner' predicate and will match the argument inside the bracket. But it will return 'false or no' since it cannot find the match.

2.Forward versus backward reasoning :

Forward reasoning(Data-driven search /forward chaining):

The **forward reasoning** is a data-driven approach. The data-Driven approach collects the facts about the problem, applies rules to the data, and produces new facts from it to move towards the goal. It is also termed forward chaining.

Forward chaining is a form of reasoning which starts 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 adds their conclusion to the known facts. This process repeats until the problem is solved.

Properties:

1. It is a bottom-up approach, as it moves from bottom to top.
2. It operates in the forward direction.
3. 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.
4. It is suitable for the planning, monitoring, control, and interpretation application. It is commonly used in an expert systems, such as CLIPS, business, and production rule systems.
5. Forward chaining reasoning applies a breadth-first search strategy.
6. It tests for all the available rules.
7. It can generate an infinite number of possible conclusions. It is aimed at any conclusion.

Backward reasoning (Goal-driven search /Backward chaining):

The **backward reasoning** is goal-driven. Goal-driven approach focus on the goal, search the facts and rules that could be able to produce that goal. It works by processing in a backward direction via rules and sub-goals to reach the facts of the problem. It is also called backward chaining. 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:

1. It is a top-down approach.
2. Backward-chaining is based on the modus ponens inference rule.
3. It operates in the backward direction.
4. In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
5. It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
6. Backward chaining reasoning applies a depth-first search strategy.
7. Backward chaining only tests for few required rules.
8. Backward-chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
9. Backward chaining is suitable for diagnostic, prescription, and debugging applications.
10. Backward chaining generates a finite number of possible conclusions.
11. Backward chaining is only aimed at the required data.

4. Symbolic reasoning under uncertainty

1.Nonmonotonic reasoning:

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

Symbolic reasoning refers to mathematical logic, more precisely first-order (predicate) logic and sometimes higher orders.

Uncertainty :

It is defined as the lack of exact information or knowledge that helps us to find the correct conclusion.

Uncertainty may be caused by problems with data such as:

1. Missing/unavailable data
2. Unreliable / ambiguous data
3. Imprecise / inconsistent representation of data
4. Guess
5. Default based

Nonmonotonic reasoning :

In Non-monotonic reasoning, some conclusions may be invalidated if we add some more information to our knowledge base. Non-monotonic reasoning deals with incomplete and uncertain models.

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 the knowledge base "**Pitty is a penguin"**, which concludes "**Pitty cannot fly**", so it invalidates the above conclusion.

Advantages :

- ❖ 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:

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

<u>Default reasoning(Logics for nonmonotonic reasoning):</u>

It is a very common form of nonmonotonic reasoning. Here we want to draw conclusions based on what is most likely to be true.

Approaches:

1. **Non-monotonic Logic :**

   The truth of the proposition may change when new information is added and logic may be built to allow the statement to be retracted. This is an extension of first-order predicate logic to include a modal operator, M. The purpose of this is to allow for consistency.

   Example:

   ∀x:plays_instrument(x) ^ M manages(x) → jazz_musician(x)

   states that, for all x is x plays an instrument and if the fact that x can manage is consistent with all other knowledge then we can conclude that x is a jazz musician.

2. **Default Logic :**

   Default logic introduces a new inference rule of the form:

   A: B

   C

   which states if A is provable and it is consistent to assume B then conclude C.

   Example:

   BIRD(x) :FLIES(x)

   FLIES(x)

2. <u>Depth first search(DFS):</u>(Refer module 2)

3. **Breadth first search(BFS):**(Refer module 2)