Jaipur Engineering College & Research Centre, Jaipur
Department of Computer Science and Engineering

Lecture Notes

Artificial Intelligence [6CS4-05]

Unit 3

Vision of the Department:

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

Mission of the Department:

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.
M2: To provide opportunities for interaction between academia and industry.
M3: To provide platform for lifelong learning by accepting the change in technologies.
M4: To develop aptitude of fulfilling social responsibilities.

Program Outcomes (PO):

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and Computer Science & Engineering specialization to the solution of complex Computer Science & Engineering problems.

2. **Problem analysis**: Identify, formulate, research literature, and analyze complex Computer Science and Engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions**: Design solutions for complex Computer Science and Engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety,and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of Computer Science and Engineering experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex Computer Science Engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional Computer Science and Engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional Computer Science and Engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the Computer Science and Engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings in Computer Science and Engineering.

10. **Communication**: Communicate effectively on complex Computer Science and Engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance**: Demonstrate knowledge and understanding of the Computer Science and Engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest contextof technological change in Computer Science and Engineering.

Program Educational Objectives (PEO):

PEO1: To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science & Engineering by way of analyzing and exploiting engineering challenges.

PEO2: To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems in Computer Science and Engineering

PEO3: To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues for Computer Science & Engineering.

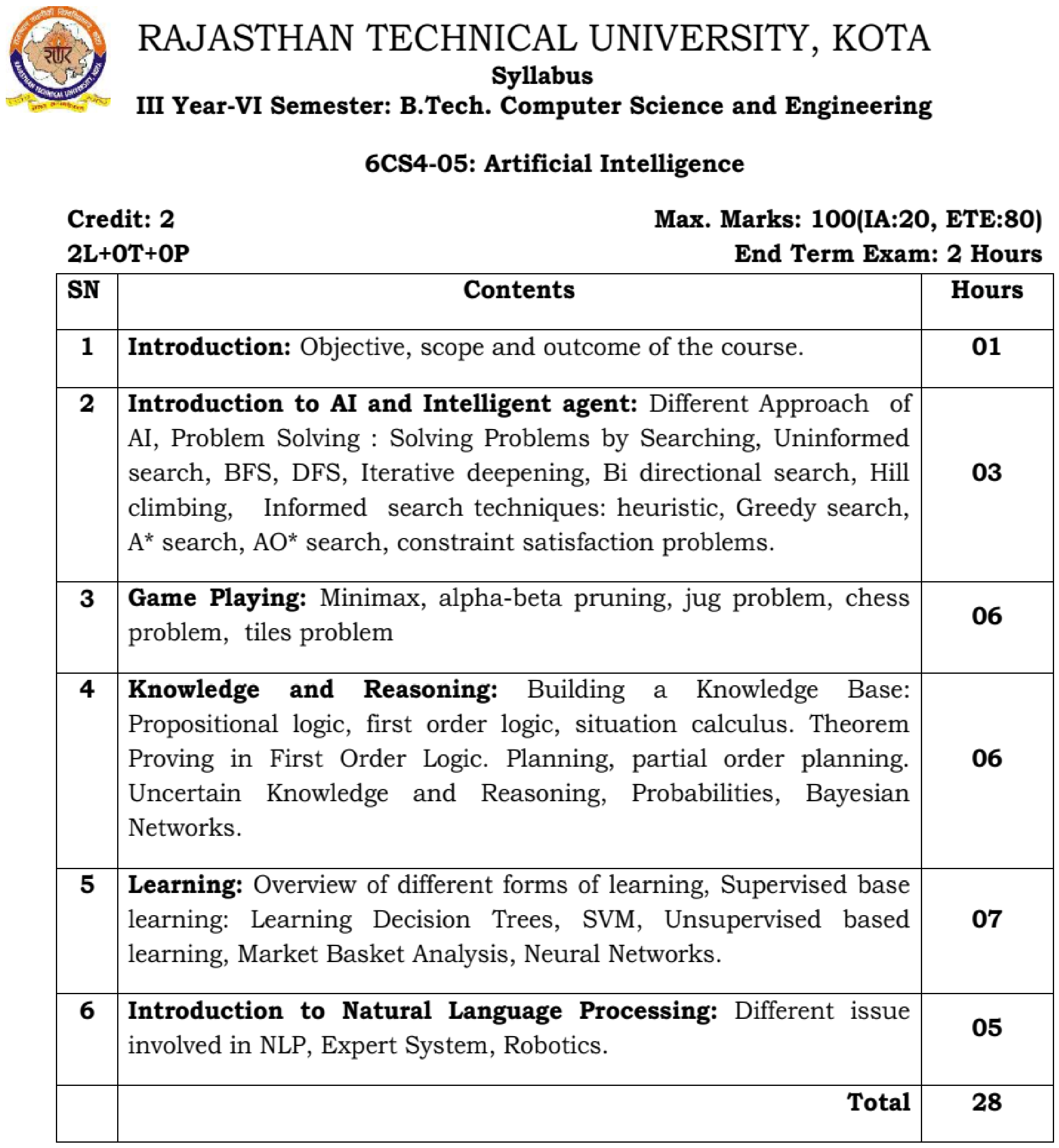
PEO4: To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self-motivated life-long learning needed for a successful professional career in Computer Science & Engineering.

PEO5: To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge in Computer Science & Engineering.

Course Outcomes (COs):

CO1: Understand the concept of Artificial Intelligence and apply various Searching techniques.
CO2: Illustrate various Game Playing in Artificial Intelligence system.
CO3: Analyze different Knowledge Representation Techniques, Neural Network, Planning, Uncertain Knowledge and Reasoning.
CO4: Apply basic concepts of Learning, Natural Language Processing, Robotics and Expert Systems in AI.

Syllabus:



RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

III Year-VI Semester: B.Tech. Computer Science and Engineering

6CS4-05: Artificial Intelligence

Credit: 2 **Max. Marks: 100(IA:20, ETE:80)**

2L+0T+0P **End Term Exam: 2 Hours**

SN	Contents	Hours
1	**Introduction:** Objective, scope and outcome of the course.	01
2	**Introduction to AI and Intelligent agent:** Different Approach of AI, Problem Solving : Solving Problems by Searching, Uninformed search, BFS, DFS, Iterative deepening, Bi directional search, Hill climbing, Informed search techniques: heuristic, Greedy search, A* search, AO* search, constraint satisfaction problems.	03
3	**Game Playing:** Minimax, alpha-beta pruning, jug problem, chess problem, tiles problem	06
4	**Knowledge and Reasoning:** Building a Knowledge Base: Propositional logic, first order logic, situation calculus. Theorem Proving in First Order Logic. Planning, partial order planning. Uncertain Knowledge and Reasoning, Probabilities, Bayesian Networks.	06
5	**Learning:** Overview of different forms of learning, Supervised base learning: Learning Decision Trees, SVM, Unsupervised based learning, Market Basket Analysis, Neural Networks.	07
6	**Introduction to Natural Language Processing:** Different issue involved in NLP, Expert System, Robotics.	05
	Total	**28**

Knowledge and Reasoning

Building a Knowledge Base

1. Prepositional Logic:

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:

a) It is Sunday.
b) The Sun rises from West (False proposition)
c) 3+3= 7(False proposition)
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.
- 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:

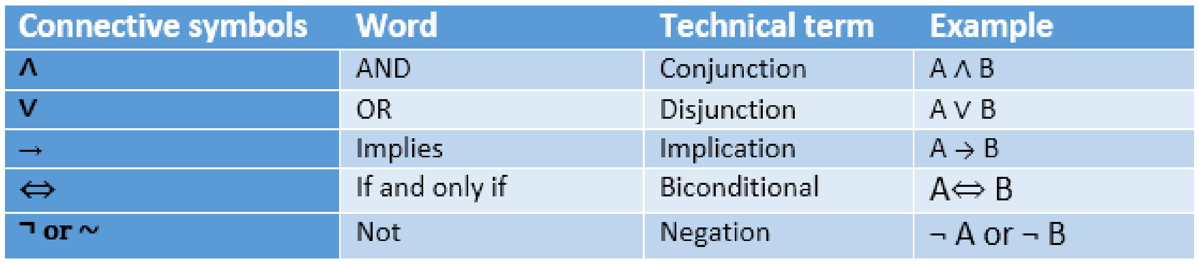
a) 2+2 is 4, it is an atomic proposition as it is a true fact.
b) "The Sun is cold" is also a proposition as it is a false fact.

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

Example:

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

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**

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	Pv Q	PvQ→¬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

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:**
  - P ∧ True = P,
  - P ∨ True= True.
- **Distributive:**
  - P∧ (Q ∨ R) = (P ∧ Q) ∨ (P ∧ R).
  - P ∨ (Q ∧ R) = (P ∨ Q) ∧ (P ∨ R).
- **DE Morgan's Law:**
  - ¬ (P ∧ Q) = (¬P) ∨ (¬Q)
  - ¬ (P ∨ Q) = (¬ P) ∧ (¬Q).
- **Double-negation elimination:**
  - ¬ (¬P) = P.

Limitations of Propositional logic:

- We cannot represent relations like ALL, some, or none with propositional logic. Example:
  a. **All the girls are intelligent.**
  b. **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.

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 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).

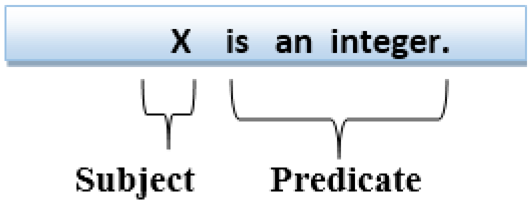
Complex Sentences:

- 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 ∀, which resembles an inverted A.

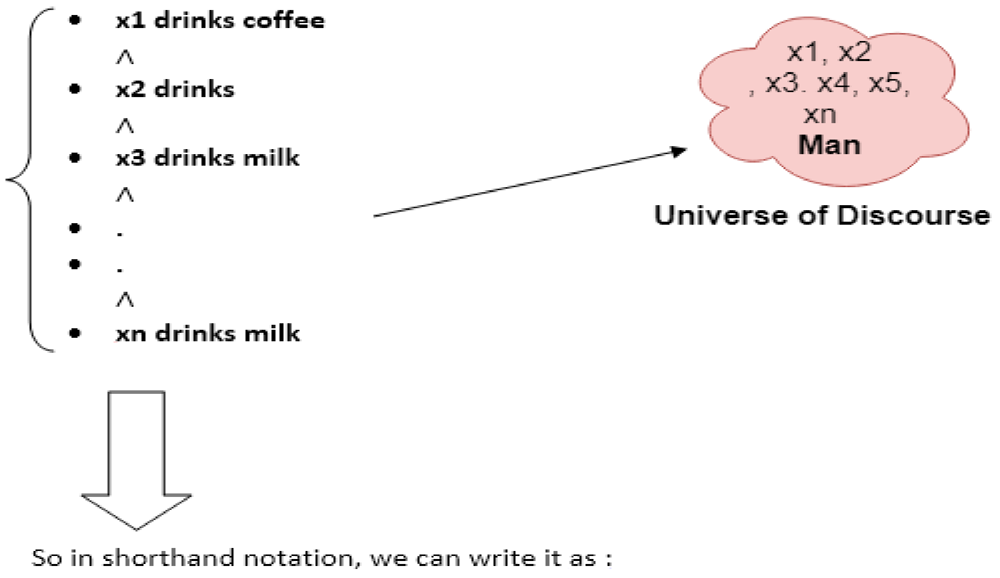
If x is a variable, then ∀x is read as:

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

Example:

All man drink coffee.

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.

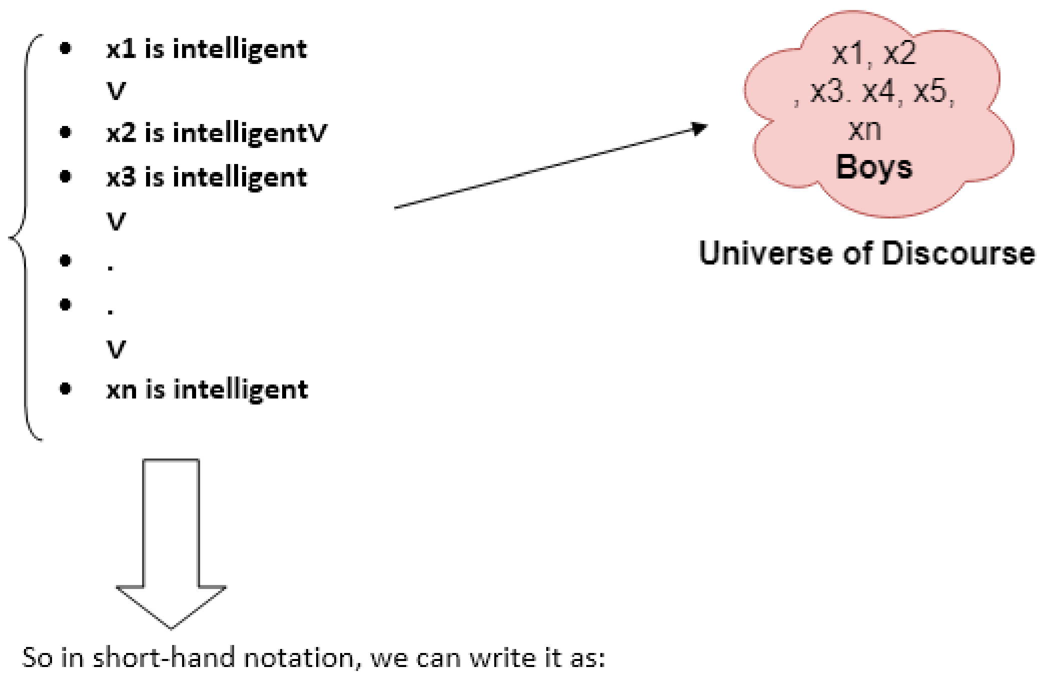
It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

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.
∀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.

Situation Calculus:

The idea behind **situation calculus** is that (reachable) states are definable in terms of the actions required to reach them. These reachable states are called situations. What is true in a situation can be defined in terms of relations with the situation as an argument. Situation calculus can be seen as a relational version of the feature-based representation of actions.

Here we only consider single agents, a fully observable environment, and deterministic actions.

Situation calculus is defined in terms of situations. A **situation** is either

- *init*, the initial situation, or
- *do(A,S),* the situation resulting from doing action *A* in situation *S*, if it is possible to do action *A* in situation *S*.

Example 14.1: Consider the domain of Figure 3.1. Suppose in the initial situation, *init*, the robot, Rob, is at location *o109* and there is a key *k1* at the mail room and a package at *storage*.

do(move(rob,o109,o103), init)

is the situation resulting from Rob moving from position *o109* in situation *init* to position *o103*. In this situation, Rob is at *o103*, the key *k1* is still at *mail*, and the package is at *storage*.

The situation

do(move(rob,o103,mail),

do(move(rob,o109,o103),

init))

is one in which the robot has moved from position *o109* to *o103* to *mail* and is currently at mail. Suppose Rob then picks up the key, *k1*. The resulting situation is

do(pickup(rob,k1),

do(move(rob,o103,mail),

do(move(rob,o109,o103),

init))).

In this situation, Rob is at position *mail* carrying the key *k1*.

A situation can be associated with a state. There are two main differences between situations and states:

- Multiple situations may refer to the same state if multiple sequences of actions lead to the same state. That is, equality between situations is not the same as equality between states.
- Not all states have corresponding situations. A state is **reachable** if a sequence of actions exists that can reach that state from the initial state. States that are not reachable do not have a corresponding situation.

Some *do(A,S)* terms do not correspond to any state. However, sometimes an agent must reason about such a (potential) situation without knowing if *A* is possible in state *S*, or if *S* is possible.
Example 14.2: The term *do(unlock(rob,door1),init)* does not denote a state at all, because it is not possible for Rob to unlock the door when Rob is not at the door and does not have the key.

A **static** relation is a relation for which the truth value does not depend on the situation; that is, its truth value is unchanging through time. A **dynamic** relation is a relation for which the truth value depends on the situation. To represent what is true in a situation, predicate symbols denoting dynamic relations have a situation argument so that the truth can depend on the situation. A predicate symbol with a situation argument is called a **fluent**.
Example 14.3: The relation *at(O,L,S)* is true when object *O* is at location *L* in situation *S*. Thus, *at* is a fluent.
The atom
at(rob,o109,init)

is true if the robot *rob* is at position *o109* in the initial situation. The atom
at(rob,o103,do(move(rob,o109,o103), init))

is *true* if robot *rob* is at position *o103* in the situation resulting from *rob* moving from position *o109* to position *o103* from the initial situation. The atom
at(k1,mail,do(move(rob,o109,o103), init))

is *true* if *k1* is at position *mail* in the situation resulting from *rob* moving from position *o109* to position *o103* from the initial situation.

A dynamic relation is axiomatized by specifying the situations in which it is true. Typically, this is done inductively in terms of the structure of situations.

- Axioms with *init* as the situation parameter are used to specify what is true in the initial situation.
- A **primitive** relation is defined by specifying when it is true in situations of the form *do(A,S)* in terms of what is true in situation *S*. That is, primitive relations are defined in terms of what is true at the previous situation.
- A **derived** relation is defined using clauses with a variable in the situation argument. The truth of a derived relation in a situation depends on what else is true in the same situation.
- Static relations are defined without reference to the situation.

Example 14.4: Suppose the delivery robot, Rob, is in the domain depicted in Figure 3.1. Rob is at location *o109*, the parcel is in the storage room, and the key is in the mail room. The following axioms describe this initial situation:
at(rob,o109,init).
at(parcel,storage,init).
at(k1,mail,init).

The *adjacent* relation is a dynamic, derived relation defined as follows:
adjacent(o109,o103,S).
adjacent(o103,o109,S).
adjacent(o109,storage,S).
adjacent(storage,o109,S).
adjacent(o109,o111,S).
adjacent(o111,o109,S).
adjacent(o103,mail,S).
adjacent(mail,o103,S).
adjacent(lab2,o109,S).
$adjacent(P_1,P_2,S) \leftarrow$
    $between(Door,P_1,P_2) \wedge$
    unlocked(Door,S).

Notice the free *S* variable; these clauses are true for all situations. We cannot omit the *S* because which rooms are adjacent depends on whether a door is unlocked. This can change from situation to situation.
The *between* relation is static and does not require a situation variable:
between(door1,o103,lab2).

We also distinguish whether or not an agent is being carried. If an object is not being carried, we say that the object is sitting at its location. We distinguish this case because an object being carried moves with the object carrying it. An object is at a location if it is sitting at that location or is being carried by an object at that location. Thus, *at* is a derived relation:
at(Ob,P,S)←
    sitting_at(Ob,P,S).

at(Ob,P,S)←
  carrying(Ob1,Ob,S)∧
  at(Ob1,P,S).

Note that this definition allows for Rob to be carrying a bag, which, in turn, is carrying a book.

The **precondition** of an action specifies when it is possible to carry out the action. The relation *poss(A,S)* is true when action *A* is possible in situation *S*. This is typically a derived relation.

Example 14.5: An agent can always put down an object it is carrying:
poss(putdown(Ag,Obj),S) ←
  carrying(Ag,Obj,S).

For the *move* action, an autonomous agent can move from its current position to an adjacent position:
poss(move(Ag,P_1,P_2),S) ←
  autonomous(Ag) ∧
  adjacent(P_1,P_2,S)∧
  sitting_at(Ag,P_1,S) .

The precondition for the unlock action is more complicated. The agent must be at the correct side of the door and carrying the appropriate key:
poss(unlock(Ag,Door),S)←
  autonomous(Ag)∧
  between(Door,P_1,P_2)∧
  at(Ag,P_1,S)∧
  opens(Key,Door)∧
  carrying(Ag,Key,S).

We do not assume that the *between* relation is symmetric. Some doors can only open one way.

We define what is true in each situation recursively in terms of the previous situation and of what action occurred between the situations. As in the feature-based representation of actions, **causal rules** specify when a relation becomes true and **frame rules** specify when a relation remains true.

Example 14.6: The primitive *unlocked* relation can be defined by specifying how different actions can affect its being true. The door is unlocked in the situation resulting from an unlock action, as long as the unlock action was possible. This is represented using the following causal rule:
unlocked(Door,do(unlock(Ag,Door),S)) ←
  poss(unlock(Ag,Door),S).

Suppose the only action to make the door locked is to lock the door. Thus, *unlocked* is true in a situation following an action if it was true before, if the action was not to lock the door, and if the action was possible:

$$
\begin{aligned}
&unlocked(Door,do(A,S)) \leftarrow\\
&\quad unlocked(Door,S) \land\\
&\quad A \neq lock(Door) \land\\
&\quad poss(A,S).
\end{aligned}
$$

This is a frame rule.

Example 14.7: The *carrying* predicate can be defined as follows.
An agent is carrying an object after picking up the object:

$$
\begin{aligned}
&carrying(Ag,Obj,do(pickup(Ag,Obj),S)) \leftarrow\\
&\quad poss(pickup(Ag,Obj),S).
\end{aligned}
$$

The only action that undoes the *carrying* predicate is the *putdown* action. Thus, *carrying* is true after an action if it was true before the action, and the action was not to put down the object. This is represented in the frame rule:

$$
\begin{aligned}
&carrying(Ag,Obj,do(A,S)) \leftarrow\\
&\quad carrying(Ag,Obj,S) \land\\
&\quad poss(A,S) \land\\
&\quad A \neq putdown(Ag,Obj).
\end{aligned}
$$

Example 14.8: The atom *sitting_at(Obj,Pos,*S_1*)* is true in a situation S_1 resulting from object *Obj* moving to *Pos*, as long as the action was possible:

$$
\begin{aligned}
&sitting\_at(Obj,Pos,do(move(Obj,Pos_0,Pos),S)) \leftarrow\\
&\quad poss(move(Obj,Pos_0,Pos),S).
\end{aligned}
$$

The other action that makes *sitting_at* true is the *putdown* action. An object is sitting at the location where the agent who put it down was located:

$$
\begin{aligned}
&sitting\_at(Obj,Pos,do(putdown(Ag,Obj),S)) \leftarrow\\
&\quad poss(putdown(Ag,Obj),S) \land\\
&\quad at(Ag,Pos,S).
\end{aligned}
$$

The only other time that *sitting_at* is true in a (non-initial) situation is when it was true in the previous situation and it was not undone by an action. The only actions that undo *sitting_at* is a *move* action or a *pickup* action. This can be specified by the following frame axiom:

$$
\begin{aligned}
&sitting\_at(Obj,Pos,do(A,S) \qquad) \leftarrow\\
&\quad poss(A,S) \land\\
&\quad sitting\_at(Obj,Pos,S) \land\\
&\quad \forall Pos_1 \qquad A \neq move(Obj,Pos,Pos_1) \land\\
&\quad \forall Ag \quad A \neq pickup(Ag,Obj) .
\end{aligned}
$$

Note that the quantification in the body is not the standard quantification for rules. This can be represented using negation as failure:

sitting_at(Obj,Pos,do(A,S) *)* ←
  poss(A,S) ∧
  sitting_at(Obj,Pos,S) ∧
  ~move_action(A,Obj,Pos) ∧
  ~pickup_action(A,Obj) .
move_action(move(Obj,Pos,Pos$_1$*),Obj,Pos).*
pickup_action(pickup(Ag,Obj),Obj).

These clauses are designed not to have a free variable in the scope of the negation.

Example 14.9: Situation calculus can represent more complicated actions than can be represented with simple addition and deletion of propositions in the state description.
Consider the *drop_everything* action in which an agent drops everything it is carrying. In situation calculus, the following axiom can be added to the definition of *sitting_at* to say that everything the agent was carrying is now on the ground:

sitting_at(Obj,Pos,do(drop_everything(Ag) *,S)* *)* ←
  poss(drop_everything(Ag),S) ∧
  at(Ag,Pos,S) ∧
  carrying(Ag,Obj,S) .

A frame axiom for *carrying* specifies that an agent is not carrying an object after a *drop_everything* action.

carrying(Ag,Obj,do(A *,S)* *)* ←
  poss(A,S) ∧
  carrying(Ag,Obj,S)∧
  A *≠drop_everything(Ag)*∧
  A ≠putdown(Ag,Obj).

The *drop_everything* action thus affects an unbounded number of objects.

Situation calculus is used for **planning** by asking for a situation in which a goal is true. Answer extraction is used to find a situation in which the goal is true. This situation can be interpreted as a sequence of actions for the agent to perform.
Example 14.10: Suppose the goal is for the robot to have the key *k1*. The following query asks for a situation where this is true:
? carrying(rob,k1,S).
This query has the following answer:

```
S=do(pickup(rob,k1),
          do(move(rob,o103,mail),
               do(move(rob,o109,o103),
                    init))).
```

The preceding answer can be interpreted as a way for Rob to get the key: it moves from *o109* to *o103*, then to *mail*, where it picks up the key.
The goal of delivering the parcel (which is, initially, in the lounge, *lng*) to *o111* can be asked with the query
? at(parcel,o111,S).

This query has the following answer:

```
S=do(move(rob, o109, o111),
          do(move(rob, lng, o109),
               do(pickup(rob, parcel),
                    do(move(rob, o109, lng), init)))).
```

Therefore, Rob should go to the lounge, pick up the parcel, go back to *o109*, and then go to *o111*.

Using the top-down proof procedure on the situation calculus definitions is very inefficient, because a frame axiom is almost always applicable. A complete proof procedure, such as iterative deepening, searches through all permutations of actions even if they are not relevant to the goal.

Theoram Proving in First Order Logic

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.

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

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

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

Example:

We can resolve two clauses which are given below:

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

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

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

[Animal (g(x) V ¬ Kills(f(x), x)].

Steps for Resolution:

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

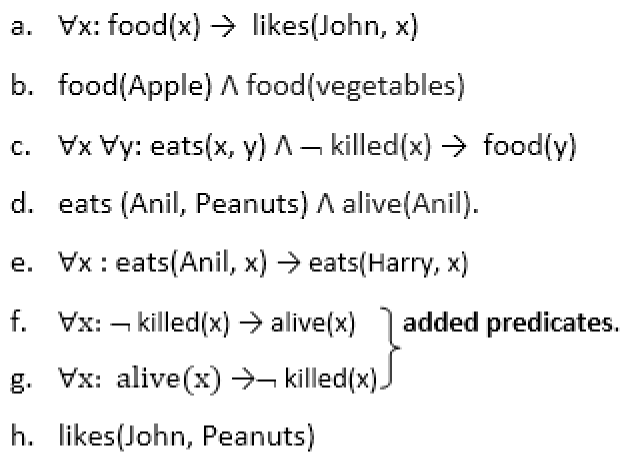
To better understand all the above steps, we will take an example in which we will apply resolution.

Example:

a. **John likes all kind of food.**
b. **Apple and vegetable are food**
c. **Anything anyone eats and not killed is food.**
d. **Anil eats peanuts and still alive**
e. **Harry eats everything that Anil eats.**
Prove by resolution that:
f. **John likes peanuts.**

Step-1: Conversion of Facts into FOL

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



a. ∀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**

. ∀x ¬ food(x) V likes(John, x)

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

- **Rename variables or standardize variables**

. ∀x ¬ food(x) V likes(John, x)

a. food(Apple) Λ food(vegetables)
b. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
c. eats (Anil, Peanuts) Λ alive(Anil)
d. ∀w¬ eats(Anil, w) V eats(Harry, w)
e. ∀g ¬killed(g)] V alive(g)
f. ∀k ¬ alive(k) V ¬ killed(k)
g. likes(John, Peanuts).
h.

- **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.

. ¬ food(x) V likes(John, x)

a. food(Apple)
b. food(vegetables)
c. ¬ eats(y, z) V killed(y) V food(z)
d. eats (Anil, Peanuts)
e. alive(Anil)

f. ¬ eats(Anil, w) V eats(Harry, w)
g. killed(g) V alive(g)
h. ¬ alive(k) V ¬ killed(k)
i. likes(John, Peanuts).

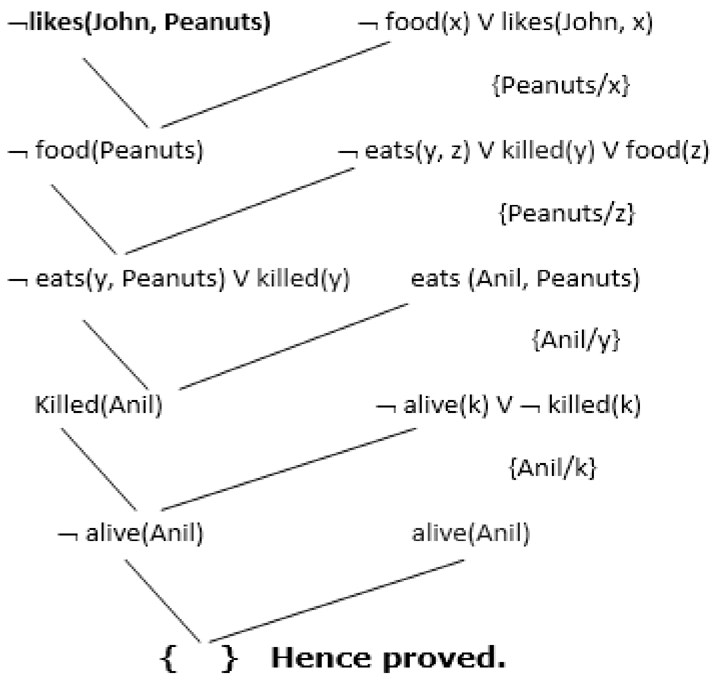
- **Distribute conjunction ∧ over disjunction ¬.** This step will not make any change in this problem.

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.

Planning: the task of coming up with a sequence of actions that will achieve a goal Search-based problem-solving agentv Logical planning agentv Complex/large scale problems?v For the discussion, we consider classicalv planning environments that are fully observable, deterministic, finite, static and discrete (in time, action, objects and effects)

Partial Order Planning

The forward and regression planners enforce a total ordering on actions at all stages of the planning process. The CSP planner commits to the particular time that the action will be carried out. This means that those planners have to commit to an ordering of actions that cannot occur concurrently when adding them to a partial plan, even if there is no particular reason to put one action before another.

The idea of a **partial-order planner** is to have a partial ordering between actions and only commit to an ordering between actions when forced. This is sometimes also called a **non-linear planner**, which is a misnomer because such planners often produce a linear plan.

A partial ordering is a less-than relation that is transitive and asymmetric. A **partial-order plan** is a set of actions together with a partial ordering, representing a "before" relation on actions, such that any total ordering of the actions, consistent with the partial ordering, will solve the goal from the initial state. Write $act_0 < act_1$ if action act_0 is before action act_1 in the partial order. This means that action act_0 must occur before action act_1.

An action, other than *start* or *finish*, will be in a partial-order plan to achieve a precondition of an action in the plan. Each precondition of an action in the plan is either true in the initial state, and so achieved by *start*, or there will be an action in the plan that achieves it.

We must ensure that the actions achieve the conditions they were assigned to achieve. Each precondition *P* of an action act_1 in a plan will have an action act_0 associated with it such that act_0 achieves precondition *P* for act_1. The triple $\langle act_0, P, act_1 \rangle$ is a **causal link**. The partial order specifies that action act_0 occurs before action act_1, which is written as $act_0 < act_1$. Any other action *A* that makes *P* false must either be before act_0 or after act_1.

Informally, a partial-order planner works as follows: Begin with the actions *start* and *finish* and the partial order $start < finish$. The planner maintains an agenda that is a set of $\langle P,A \rangle$ pairs, where *A* is an action in the plan and *P* is an atom that is a precondition of *A* that must be achieved. Initially the agenda contains pairs $\langle G,finish \rangle$, where *G* is an atom that must be true in the goal state.

At each stage in the planning process, a pair $\langle G,act_1 \rangle$ is selected from the agenda, where *P* is a precondition for action act_1. Then an action, act_0, is chosen to achieve *P*. That action is either already in the plan - it could be the *start* action, for example - or it is a new action that is added to the plan. Action act_0 must happen before act_1 in the partial order. It adds a causal link that records that act_0 achieves *P* for action act_1. Any action in the plan that deletes *P* must happen either before act_0 or after act_1. If act_0 is a new action, its preconditions are added to the agenda, and the process continues until the agenda is empty.

This is a non-deterministic procedure. The "choose" and the "either ...or ..." form choices that must be searched over. There are two choices that require search:

- which action is selected to achieve *G* and
- whether an action that deletes *G* happens before act_0 or after act_1.
- **non-deterministic procedure**

Uncertain Knowledge and Reasoning

Uncertainty:

Till now, we have learned knowledge representation using first-order logic and propositional logic with certainty, which means we were sure about the predicates. With this knowledge representation, we might write A→B, which means if A is true then B is true, but consider a situation where we are not sure about whether A is true or not then we cannot express this statement, this situation is called uncertainty.

So to represent uncertain knowledge, where we are not sure about the predicates, we need uncertain reasoning or probabilistic reasoning.

Following are some leading causes of uncertainty to occur in the real world.

1. Information occurred from unreliable sources.
2. Experimental Errors
3. Equipment fault
4. Temperature variation
5. Climate change.

Probabilistic reasoning:

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge. In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty.

We use probability in probabilistic reasoning because it provides a way to handle the uncertainty that is the result of someone's laziness and ignorance.

In the real world, there are lots of scenarios, where the certainty of something is not confirmed, such as "It will rain today," "behavior of someone for some situations," "A match between two teams or two players." These are probable sentences for which we can assume that it will happen but not sure about it, so here we use probabilistic reasoning.

Need of probabilistic reasoning in AI:

- When there are unpredictable outcomes.
- When specifications or possibilities of predicates becomes too large to handle.
- When an unknown error occurs during an experiment.

In probabilistic reasoning, there are two ways to solve problems with uncertain knowledge:

- **Bayes' rule**
- **Bayesian Statistics**
- As probabilistic reasoning uses probability and related terms, so before understanding probabilistic reasoning, let's understand some common terms:
- **Probability:** Probability can be defined as a chance that an uncertain event will occur. It is the numerical measure of the likelihood that an event will occur. The value of probability always remains between 0 and 1 that represent ideal uncertainties.

$0 \leq P(A) \leq 1$, where P(A) is the probability of an event A.

$P(A) = 0$, indicates total uncertainty in an event A.

P(A) =1, indicates total certainty in an event A.

We can find the probability of an uncertain event by using the below formula.

$$\textbf{Probability of occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

- P(¬A) = probability of a not happening event.
- P(¬A) + P(A) = 1.

Event: Each possible outcome of a variable is called an event.

Sample space: The collection of all possible events is called sample space.

Random variables: Random variables are used to represent the events and objects in the real world.

Prior probability: The prior probability of an event is probability computed before observing new information.

Posterior Probability: The probability that is calculated after all evidence or information has taken into account. It is a combination of prior probability and new information.

Conditional probability:

Conditional probability is a probability of occurring an event when another event has already happened.

Let's suppose, we want to calculate the event A when event B has already occurred, "the probability of A under the conditions of B", it can be written as:

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

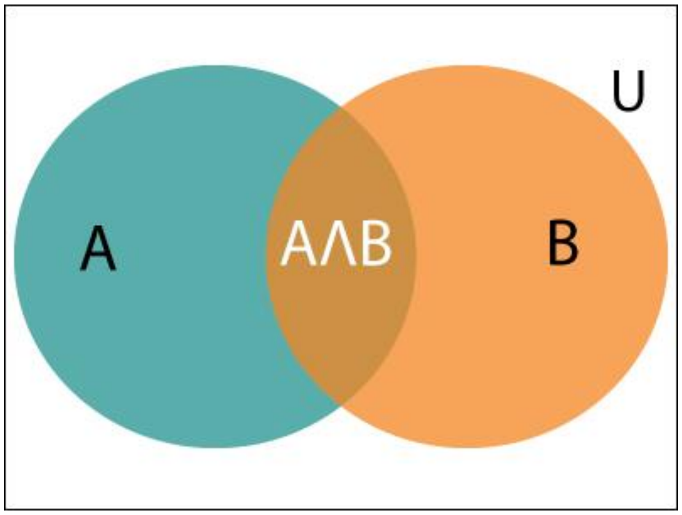
Where P($A \wedge B$)= Joint probability of a and B

P(B)= Marginal probability of B.

If the probability of A is given and we need to find the probability of B, then it will be given as:

$$P(B|A) = \frac{P(A \wedge B)}{P(A)}$$

It can be explained by using the below Venn diagram, where B is occurred event, so sample space will be reduced to set B, and now we can only calculate event A when event B is already occurred by dividing the probability of **P(A∧*B*) by P(B)**.



Example:

In a class, there are 70% of the students who like English and 40% of the students who likes English and mathematics, and then what is the percent of students those who like English also like mathematics?

Solution:
Let, A is an event that a student likes Mathematics
B is an event that a student likes English.
Hence, 57% are the students who like English also like Mathematics.

Bayesian Networks

Bayesian belief network is key computer technology for dealing with probabilistic events and to solve a problem which has uncertainty. We can define a Bayesian network as:

"A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph."

It is also called a **Bayes network, belief network, decision network**, or **Bayesian model**.

Bayesian networks are probabilistic, because these networks are built from a **probability distribution**, and also use probability theory for prediction and anomaly detection.

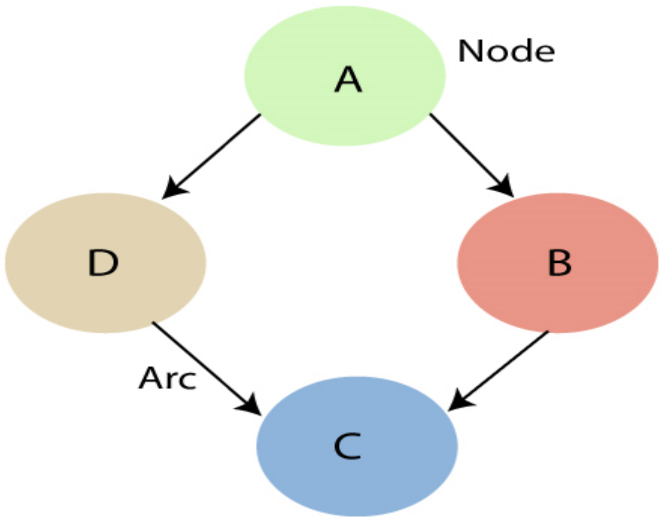
Real world applications are probabilistic in nature, and to represent the relationship between multiple events, we need a Bayesian network. It can also be used in various tasks including **prediction, anomaly detection, diagnostics, automated insight, reasoning, time series prediction**, and **decision making under uncertainty**.

Bayesian Network can be used for building models from data and experts opinions, and it consists of two parts:

- **Directed Acyclic Graph**
- **Table of conditional probabilities.**

The generalized form of Bayesian network that represents and solve decision problems under uncertain knowledge is known as an **Influence diagram**.

A Bayesian network graph is made up of nodes and Arcs (directed links), where:



- Each **node** corresponds to the random variables, and a variable can be **continuous** or **discrete**.
- **Arc or directed arrows** represent the causal relationship or conditional probabilities between random variables. These directed links or arrows connect the pair of nodes in the graph.
  These links represent that one node directly influence the other node, and if there is no directed link that means that nodes are independent with each other

- **In the above diagram, A, B, C, and D are random variables represented by the nodes of the network graph.**
- **If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B.**
- **Node C is independent of node A.**

The Bayesian network has mainly two components:

- **Causal Component**
- **Actual numbers**

Each node in the Bayesian network has condition probability distribution **$P(X_i |Parent(X_i))$**, which determines the effect of the parent on that node.

Bayesian network is based on Joint probability distribution and conditional probability. So let's first understand the joint probability distribution:

Joint probability distribution:

If we have variables x1, x2, x3,....., xn, then the probabilities of a different combination of x1, x2, x3.. xn, are known as Joint probability distribution.

$P[x_1, x_2, x_3,....., x_n]$, it can be written as the following way in terms of the joint probability distribution.

$= P[x_1| x_2, x_3,....., x_n]P[x_2, x_3,....., x_n]$

$= P[x_1| x_2, x_3,....., x_n]P[x_2|x_3,....., x_n]....P[x_{n-1}|x_n]P[x_n]$.

In general for each variable Xi, we can write the equation as:

$$P(X_i|X_{i-1},........., X_1) = P(X_i |Parents(X_i))$$

Explanation of Bayesian network:

Let's understand the Bayesian network through an example by creating a directed acyclic graph:

Example: Harry installed a new burglar alarm at his home to detect burglary. The alarm reliably responds at detecting a burglary but also responds for minor earthquakes. Harry has two neighbors David and Sophia, who have taken a responsibility to inform Harry at work when they hear the alarm. David always calls Harry when he hears the alarm, but sometimes he got confused with the phone ringing and calls at that time too. On the other hand, Sophia likes to listen to high music, so sometimes she misses to hear the alarm. Here we would like to compute the probability of Burglary Alarm.

Problem:

Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and David and Sophia both called the Harry.

Solution:

- The Bayesian network for the above problem is given below. The network structure is showing that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Sophia's calls depend on alarm probability.
- The network is representing that our assumptions do not directly perceive the burglary and also do not notice the minor earthquake, and they also not confer before calling.
- The conditional distributions for each node are given as conditional probabilities table or CPT.
- Each row in the CPT must be sum to 1 because all the entries in the table represent an exhaustive set of cases for the variable.
- In CPT, a boolean variable with k boolean parents contains 2^K probabilities. Hence, if there are two parents, then CPT will contain 4 probability values

List of all events occurring in this network:

- **Burglary (B)**
- **Earthquake(E)**
- **Alarm(A)**
- **David Calls(D)**
- **Sophia calls(S)**

We can write the events of problem statement in the form of probability: **P[D, S, A, B, E]**, can rewrite the above probability statement using joint probability distribution:

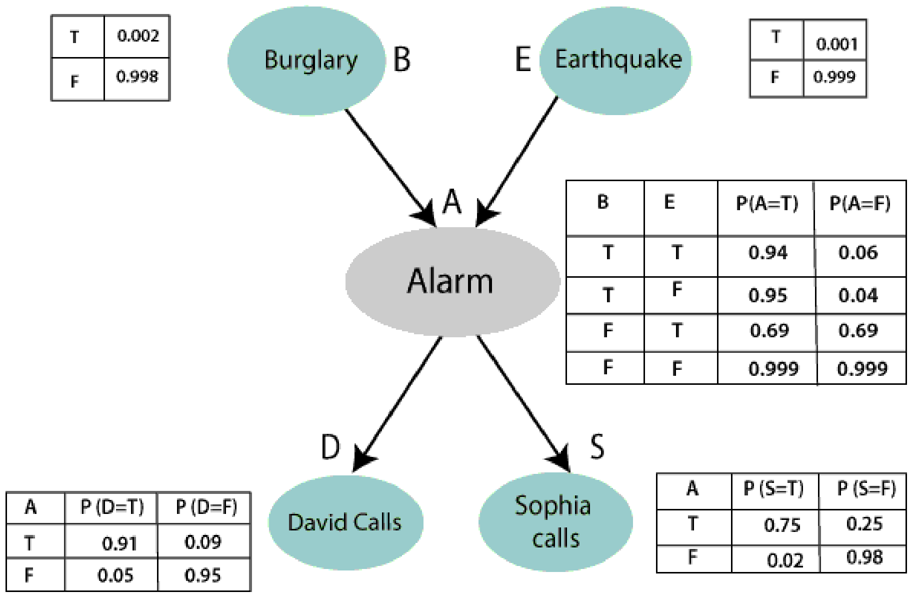
P[D, S, A, B, E]= P[D | S, A, B, E]. P[S, A, B, E]

=P[D | S, A, B, E]. P[S | A, B, E]. P[A, B, E]

= P [D| A]. P [S| A, B, E]. P[A, B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B |E]. P[E]



T	0.002
F	0.998

T	0.001
F	0.999

B	E	P(A=T)	P(A=F)
T	T	0.94	0.06
T	F	0.95	0.04
F	T	0.69	0.69
F	F	0.999	0.999

A	P (D=T)	P (D=F)
T	0.91	0.09
F	0.05	0.95

A	P (S=T)	P (S=F)
T	0.75	0.25
F	0.02	0.98

Let's take the observed probability for the Burglary and earthquake component:
P(B= True) = 0.002, which is the probability of burglary.
P(B= False)= 0.998, which is the probability of no burglary.
P(E= True)= 0.001, which is the probability of a minor earthquake
P(E= False)= 0.999, Which is the probability that an earthquake not occurred.
We can provide the conditional probabilities as per the below tables:

Conditional probability table for Alarm A:
The Conditional probability of Alarm A depends on Burglar and earthquake:

B	E	P(A= True)	P(A= False)
True	True	0.94	0.06
True	False	0.95	0.04
False	True	0.31	0.69
False	False	0.001	0.999

Conditional probability table for David Calls:

The Conditional probability of David that he will call depends on the probability of Alarm.

A	P(D= True)	P(D= False)
True	0.91	0.09
False	0.05	0.95

Conditional probability table for Sophia Calls:

The Conditional probability of Sophia that she calls is depending on its Parent Node "Alarm."

A	P(S= True)	P(S= False)
True	0.75	0.25
False	0.02	0.98

From the formula of joint distribution, we can write the problem statement in the form of probability distribution:

P(S, D, A, ¬B, ¬E) = P (S|A) *P (D|A)*P (A|¬B ^ ¬E) *P (¬B) *P (¬E).

= 0.75* 0.91* 0.001* 0.998*0.999

= 0.00068045.

Hence, a Bayesian network can answer any query about the domain by using Joint distribution.

The semantics of Bayesian Network:

There are two ways to understand the semantics of the Bayesian network, which is given below:

1. To understand the network as the representation of the Joint probability distribution.

It is helpful to understand how to construct the network.

2. To understand the network as an encoding of a collection of conditional independence statements.