**Module-2**

1. **Logical Agents: Knowledge representation structures:**

The different structures to represent knowledge are given below

1. **Logical Representation**

Logical representation means drawing a conclusion based on various conditions. This representation lays down some important communication rules. It consists of precisely defined syntax and semantics which supports the sound inference. Each sentence can be translated into logics using syntax and semantics.

* **Syntax:**
* Syntaxes are the rules which decide how we can construct legal sentences in the logic.
* It determines which symbol we can use in knowledge representation.
* How to write those symbols**.**
* **Semantics:**
* Semantics are the rules by which we can interpret the sentence in the logic.
* Semantic also involves assigning a meaning to each sentence.

**Logical representation can be categorised into mainly two logics:**

* Propositional Logics
* Predicate logics
* **Advantages of logical representation:**

1. Logical representation enables us to do logical reasoning.
2. Logical representation is the basis for the programming languages.

* **Disadvantages of logical Representation:**

1. Logical representations have some restrictions and are challenging to work with.
2. Logical representation technique may not be very natural, and inference may not be so efficient.
3. **Semantic Network Representation**

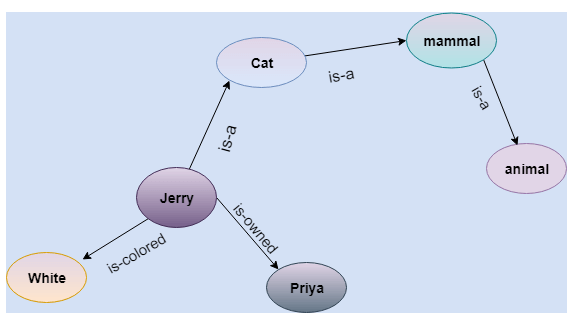
In Semantic networks, we can represent our knowledge in the form of graphical networks. This network consists of nodes representing objects and arcs which describe the relationship between those objects. Semantic networks can categorize the object in different forms and can also link those objects. Semantic networks are easy to understand and can be easily extended.

This representation consist of mainly two types of relations:

* 1. IS-A relation (Inheritance)
  2. Kind-of-relation

**Example:** Following are some statements which we need to represent in the form of nodes and arcs.

Statements:

1. Jerry is a cat.
2. Jerry is a mammal
3. Jerry is owned by Priya.
4. Jerry is brown colored.
5. All Mammals are animal.

In the above diagram, the different type of knowledge in the form of nodes and arcs are registered. Each object is connected with another object by some relation.

* **Drawbacks in Semantic representation:**

1. Semantic networks take more computational time at runtime as we need to traverse the complete network tree to answer some questions. It might be possible in the worst case scenario that after traversing the entire tree, we find that the solution does not exist in this network.
2. Semantic networks try to model human-like memory (Which has 1015 neurons and links) to store the information, but in practice, it is not possible to build such a vast semantic network.
3. These types of representations are inadequate as they do not have any equivalent quantifier, e.g., for all, for some, none, etc.
4. Semantic networks do not have any standard definition for the link names.
5. These networks are not intelligent and depend on the creator of the system.

* **Advantages of Semantic network:**

1. Semantic networks are a natural representation of knowledge.
2. Semantic networks convey meaning in a transparent manner.
3. These networks are simple and easily understandable.
4. **Frame Representation**

A frame is a record like structure which consists of a collection of attributes and its values to describe an entity in the world. Frames are the AI data structure which divides knowledge into substructures by representing stereotypes situations. It consists of a collection of slots and slot values. These slots may be of any type and sizes. Slots have names and values which are called facets. A frame is also known as **slot-filter knowledge representation** in artificial intelligence.

**Example:**

Let's suppose we are taking an entity, Peter. Peter is an engineer as a profession, and his age is 25, he lives in city London, and the country is England. So following is the frame representation for this:

|  |  |
| --- | --- |
| **Slots** | **Filter** |
| Name | Peter |
| Profession | Doctor |
| Age | 25 |
| Marital Status | Single |
| Weight | 78 |

* **Advantages of frame representation:**

1. The frame knowledge representation makes the programming easier by grouping the related data.
2. The frame representation is comparably flexible and used by many applications in AI.
3. It is very easy to add slots for new attribute and relations.
4. It is easy to include default data and to search for missing values.
5. Frame representation is easy to understand and visualize.

* **Disadvantages of frame representation:**

1. In frame system inference mechanism is not be easily processed.
2. Inference mechanism cannot be smoothly proceeded by frame representation.
3. Frame representation has a much generalized approach.
4. **Scripts**

A script is a structure that prescribes a set of circumstances that could be expected to follow on from one another.

Scripts are useful because

* 1. Events tend to occur in known runs or patterns.
  2. Entry conditions exist which allow an event to take place.
  3. Casual relationships between events exist.
  4. Prerequisites exist.
* **Components of a script**

The components of a script include:

* **Entry condition:** These are basic condition which must be fulfilled before events in the script can occur.
* **Results:**Condition that will be true after events in script occurred.
* **Props:**Slots representing objects involved in events
* **Roles:**These are the actions that the individual participants perform.
* **Track:**Variations on the script. Different tracks may share components of the same scripts.
* **Scenes:**The sequence of events that occur.
* **Advantages of Scripts**
* Ability to predict events.
* A single coherent interpretation maybe builds up from a collection of observations.
* **Disadvantages of Scripts**
* Less general than frames.
* May not be suitable to represent all kinds of knowledge

1. **Logic:**

* Sentence are expressed according to the syntax of the representation language, which specifies all the sentences that are well formed. Example: x+y=4.
* A logic must define the **semantics** or meaning of sentences. The semantics definesthe **truth** of each sentence with respect to each **possible world**. Example: “x + y =4” is true in a world where x is 2 and yis 2, but false in a world where x is 1 and y is 1.
* In standard logics, every sentence must beeither true or false in each possible world—there is no “in between.
* When we need to be precise, we use the term **model** in place of “possible world.”
* Models are mathematical abstractions, each of which simply fixesthe truth or falsehood of every relevant sentence.
* If a sentence **α** is true in model m, we say that m **satisfies α** or sometimes m **is a model of α**. We use the notation **M(α)** to mean the set of all models of **α.**
* Logical reasoning involves the relation of logical **entailment** between sentences—the idea that a sentence *followslogically* from another sentence. In mathematical notation, we write

**α |= β means that α entails the sentence β**

* The formal definition of entailment is this:α |= β if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write.

**α |= β if and only if M(α) ⊆ M(β)** .

example :x = 0 entails the sentence xy = 0.

* An inference algorithm that derives only entailed sentences is called **sound** or **truthpreserving.** Soundness is a highly desirable property
* The property of **completeness** is also desirable: an inference algorithm is complete ifit can derive any sentence that is entailed.
* **Grounding**—the connection between logical reasoningprocesses and the real environment in which the agent exists.

****

Figure 1: relationship

Sentences are physical configurations of the agent, and reasoning is a processof constructing new physical configurations from old ones. Logical reasoning should make sure that the new configurations represent aspects of the world that actually follow from theaspects that the old configurations represent.

1. **Prepositional Logic: A very Simple 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:**

* It is Sunday.
* The Sun rises from West (False proposition)
* 3+3= 7(False proposition)
* 5 is a prime number.
* 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.

1. **Syntax:**

* The syntax of propositional logic defines the allowable sentences.
* There are two types of Propositions:
* **Atomic Propositions**
* **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. 2+2 is 4, it is an atomic proposition as it is a **true** fact.
2. "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. "It is raining today, and street is wet."
2. "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:

* **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
* **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.

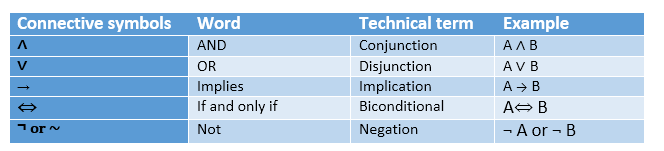
* **Disjunction:** A sentence which has ∨ connective, such as **P ∨ Q**. is called disjunction, where P and Q are the propositions.

Example: "Ritika is a doctor or Engineer",  
Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as **P ∨ Q**.

* **Implication:** A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as  
   Example: **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
* **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

:

****

Figure 2: A BNF (Backus–Naur Form) grammar of sentences in propositional logic along with operator precedences, from highest to lowest.

1. **Semantics**

* The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply fixes the **truth value**—true or false—for every proposition symbol.
* For example, if the sentences in the knowledge base make use of the proposition symbols α, β, γ
* The semantics for propositional logic must specify how to compute the truth value of

*any*sentence, given a model.

* Semantics for Atomic sentences are
* True is true in every model and False is false in every model.
* The truth value of every other proposition symbol must be specified directly in themodel. Example: in model m is false.
* For complex sentences, we have five rules, which hold for any subsentences P and Q in anymodel m (here “iff” means “if and only if”):
* ￢P is true iff P is false in m.
* P ∧ Q is true iff both P and Q are true in m.
* P ∨ Q is true iff either P or Q is true in m.
* P ⇒ Q is true unless P is true and Q is false in m.
* P ⇔ Q is true iff P and Q are both true or both false in m.
* The rules can also be expressed with **truth tables** that specify the truth value of a complex sentence for each possible assignment of truth values to its components.



Figure 3:Truth tables for the five logical connectives

1. **Logical equivalence**

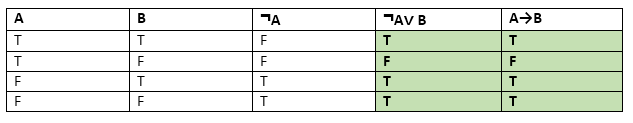
* Logical equivalence is one of the features of propositional logic. Two propositions are said to be logically equivalent if and only if the columns in the truth table are identical to each other.
* Let's take two propositions A and B, so for logical equivalence, we can write it as A⇔B. In below truth table we can see that column for ¬A∨ B and A→B, are identical hence A is Equivalent to B



Figure 4:Standard logical equivalences. the symbols stand for any arbitrary sentences of prepositional logics

1. **Propositional Theorem Proving**

* A different approach to using logic to solve problems is to use logical rules of inference to generate logical implications in some cases, this can be less work than model-checking (i.e. generating a truth table).
* **Logical Equivalence:** Two sentences *α* and *β* are logically equivalent if they are true in the same set of models. Example: P ∧Q and Q∨P are logically equivalent. Two sentences α and β are equivalentonly if each of them entails the other:

**α≡βif and only if α|= βand β|= α.**

* **Validity:**A sentence is valid if it is true in all models.Valid sentences are also known as **tautologies**—they are necessarily true. Every valid sentence is logically equivalent to *True*.
* **Deduction theorem:**

***For any sentences* α*and* β*,* α|= β*if and only if the sentence* (α⇒β) *is valid.***

* **Satisfiability:** A sentence is satisfiable if it is true in, or satisfied by, *some* model.Satisfiability can be checked by enumerating the possible models until one isfound that satisfies the sentence.Validity and satisfiability are connected:
* α is valid iff￢α is unsatisfiable;
* α is satisfiableiff￢α is not valid
* α |= β*if and only if the sentence* (α∧￢β) *is unsatisfiable.*

1. **Inference and Proofs**

Inferences rules (such as Modus Ponens and And-Elimination) can be applied to derived to a proof.

A,A=>B

---------modusponens

B

**Modus Ponens:**This rule says that if you are given a sentence A, and a sentence A => B, you may infer B.

A∧B

-----**and**-elimination

A

A∧B

-----

B

**And Elimination**: these two rules encode the (obvious!) fact that if the sentence A∧ B is true, then A is true, and also B is true. From A ∧ B we can infer B or A.

The equivalence for biconditional elimination yields the two inference rules

and

* **Monotonicity**: Which says that the set of entailedsentences can only *increase* as information is added to the knowledge base. For any sentences α and β,

ifKB |= α then KB ∧β |= α .

Monotonicity means that inference rules can be applied whenever suitable premises are found in the knowledge base, what else in the knowledge base cannot invalidate any conclusion already inferred.

1. **Proof by Resolution**

* **Resolution:** An inference rule that yields a complete inference algorithm when coupled with any complete search algorithm.
* **Clause:** A disjunction of literals. (e.g. A∨B). A single literal can be viewed as a **unit clause** (a disjunction of one literal ).
* **Unit resolution inference rule:**Takes a clause and a literal and produces a new clause.
* **Full resolution rule:**Takes 2 clauses and produces a new clause.

where *li* and mj are complementary literals.

The resolution algorithm is sound and complete.



Figure 7:A simple resolution algorithm for propositional logic.

1. **Conjuctive Normal Form**

* **Conjunctive normal form (CNF):**A sentence expressed as a conjunction of clauses is said to be in CNF.
* Every sentence of propositional logic is logically equivalent to a conjunction of clauses, after converting a sentence into CNF, it can be used as input to a resolution procedure.
* **Procedure for converting a sentence to CNF is described below.**

**Consider a sentence C**

1. Use biconditional elimination
2. Use implication elimination
3. Apply Demorgan’s law
4. Apply distributive law

The original sentence is in CNF

1. **Horn Clauses and Definite Clauses**

* **Definite clause:** A disjunction of literals of which exactly one is positive.

(e.g. ¬ L1,1∨¬Breeze∨B1,1)

* Every definite clause can be written as an implication, whose premise is a conjunction of positive literals and whose conclusion is a single positive literal.
* **Horn clause:**A disjunction of literals of which at most one is positive. (All definite clauses are Horn clauses.)
* In Horn form, the premise is called the **body** and the conclusion is called the **head**.

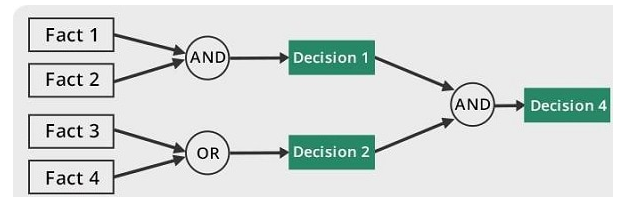
A sentence consisting of a single positive literal is called a **fact**, it too can be written in implication form.

* Horn clause are closed under resolution: if you resolve 2 horn clauses, you get back a horn clause.
* Inference with horn clauses can be done through the forward-chaining and backward-chaining algorithms.
* Deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base.
* **Goal clause:** A clause with no positive literals.

****

Figure 8:A grammar for conjunctive normal form, Horn clauses, and definite clauses.

1. **Forward and Backward Chaining**

* **Forward Chaining**
* Forward chaining is a method of reasoning in artificial intelligence in which inference rules are applied to existing data to extract additional data until an endpoint (goal) is achieved.
* In this type of chaining, the inference engine starts by evaluating existing facts, derivations, and conditions before deducing new information. An endpoint (goal) is achieved through the manipulation of knowledge that exists in the knowledge base.
* Forward chaining can be used in planning, monitoring, controlling, and interpreting applications.

#### Properties of forward chaining

* The process uses a down-up approach (bottom to top).
* It starts from an initial state and uses facts to make a conclusion.
* This approach is data-driven.
* It’s employed in expert systems and production rule system.

#### Examples of forward chaining

A simple example of forward chaining can be explained in the following sequence.

A

A🡪B

B

A is the starting point. A🡪B represents a fact. This fact is used to achieve a decision B.

A practical example will go as follows;

Tom is running (A)

If a person is running, he will sweat (A->B)

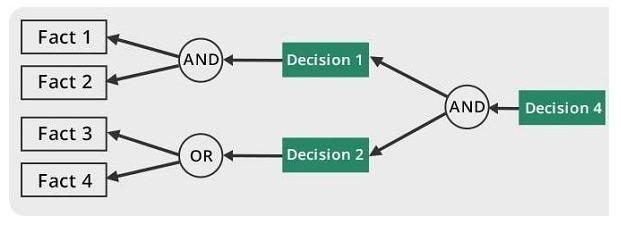
Therefore, Tom is sweating. (B)

#### Advantages

* It can be used to draw multiple conclusions.
* It provides a good basis for arriving at conclusions.
* It’s more flexible than backward chaining because it does not have a limitation on the data derived from it.

#### Disadvantages

* The process of forward chaining may be time-consuming. It may take a lot of time to eliminate and synchronize available data.
* Unlike backward chaining, the explanation of facts or observations for this type of chaining is not very clear. The former uses a goal-driven method that arrives at conclusions efficiently.
* **Backward chaining**
* Backward chaining is a concept in artificial intelligence that involves backtracking from the endpoint or goal to steps that led to the endpoint. This type of chaining starts from the goal and moves backward to comprehend the steps that were taken to attain this goal.
* The backtracking process can also enable a person establish logical steps that can be used to find other important solutions.



* Backward chaining can be used in debugging, diagnostics, and prescription applications.

#### Properties of backward chaining

* The process uses an up-down approach (top to bottom).
* It’s a goal-driven method of reasoning.
* The endpoint (goal) is subdivided into sub-goals to prove the truth of facts.
* A backward chaining algorithm is employed in inference engines, game theories, and complex database systems.
* The [modus ponens inference rule](https://en.wikipedia.org/wiki/Modus_ponens) is used as the basis for the backward chaining process. This rule states that if both the conditional statement (p->q) and the antecedent (p) are true, then we can infer the subsequent (q).

#### Example of backward chaining

#### The information provided in the previous example (forward chaining) can be used to provide a simple explanation of backward chaining. Backward chaining can be explained in the following sequence.

B

A->B

A

B is the goal or endpoint, that is used as the starting point for backward tracking. A is the initial state. A->B is a fact that must be asserted to arrive at the endpoint B.

A practical example of backward chaining will go as follows:

Tom is sweating (B).

If a person is running, he will sweat (A->B).

Tom is running (A)

**Differences between Forward and Backward Chaining**

|  |  |  |
| --- | --- | --- |
| Sl.No | Forward Chaining | Backward Chaining |
|  | Forward chaining starts from known facts and applies inference rule to extract more data unit it reaches to the goal. | Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal. |
|  | It is a bottom-up approach | It is a top-down approach |
|  | Forward chaining is known as data-driven inference technique as we reach to the goal using the available data. | Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts. |
|  | Forward chaining reasoning applies a breadth-first search strategy. | Backward chaining reasoning applies a depth-first search strategy. |
|  | Forward chaining tests for all the available rules | Backward chaining only tests for few required rules. |
|  | Forward chaining is suitable for the planning, monitoring, control, and interpretation application. | Backward chaining is suitable for diagnostic, prescription, and debugging application. |
|  | Forward chaining can generate an infinite number of possible conclusions. | Backward chaining generates a finite number of possible conclusions. |
|  | It operates in the forward direction. | It operates in the backward direction. |
|  | Forward chaining is aimed for any conclusion. | Backward chaining is only aimed for the required data |

1. **A complete backtracking algorithm**

Davis–Putnam algorithm(DPLL) provides three improvements over other algorithms

* **Early termination:** The algorithm detects whether the sentence must be true or false,

even with a partially completed model. A clause is true if any literal is true, even if

the other literals do not yet have truth values; hence, the sentence as a whole could be

judged true even before the model is complete. For example, the sentence (A ∨ B) ∧

(A ∨ C) is true if A is true, regardless of the values of B and C

* **Pure symbol heuristic:** A **pure symbol** is a symbol that always appears with the same

“sign” in all clauses. For example, in the three clauses (A ∨￢B), (￢B ∨￢C), and

(C ∨ A), the symbol A is pure because only the positive literal appears, B is pure

because only the negative literal appears, and C is impure.

* **Unit clause heuristic**: A **unit clause** was defined earlier as a clause with just one literal.In the context of DPLL, it also means clauses in which all literals but one are

already assigned false by the model. For example, if the model contains B =true,

then (￢B ∨￢C) simplifies to ￢C, which is a unit clause.

* The DPLL algorithm is shown in Figure



Figure 11:The DPLL algorithm for checking satisfiability of a sentence in propositional

* Some of the features of DPLL algorithm is listed below

1. **Component analysis** As DPLL assigns truth valuesto variables, the set of clauses may become separated into disjoint subsets, called **components**, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.
2. **Variable and value ordering**: Our simple implementation of DPLL uses an arbitrary variable ordering and always tries the value *true* before *false*
3. **Intelligent backtracking** Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict.
4. **Random restarts** Sometimes a run appears not to be making progress. After restarting, different random choices (in variable and value selection) are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.
5. **Clever indexing** :The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things as “the set of clauses in which variable Xi appears as a positive literal.” the indexing structures must be updated dynamically as the computation proceeds.
6. **First order logic**

* First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
* 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 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, ...
* First-order logic also has two main parts:
* Syntax
* 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:

|  |  |
| --- | --- |
| **Name** | **Symbol** |
| **Constant** | 1, 6, A,W,New York, Elie, Dog... |
| **Variables** | a, b, c, x, y, z... |
| **Predicates** | <, >, brother, sister, father... |
| **Equality** | == |
| **Function** | Sqrt, LessThan, Sin(θ),LeftLegOf... |
| **Quantifier** | ∀, ∃ |
| **Connectives** | ∧, ∨, ¬, ⇒, ⇔ |

**Atomic sentences:**

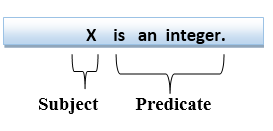
* Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
* We can represent atomic sentences as **Predicate (term1, term2, ......, term n)**.

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

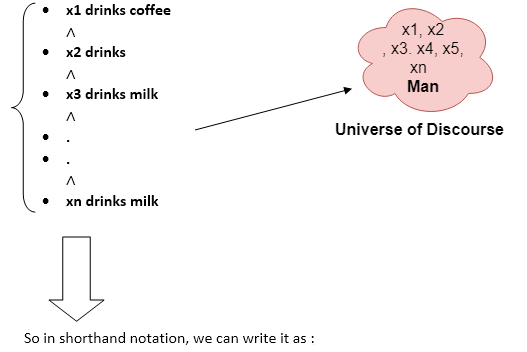
The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.

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

* **For all x**
* **For each x**
* **For every x.**
* **In universal quantifier we use implication "→".**

Example:

**All man drink coffee**



**∀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.

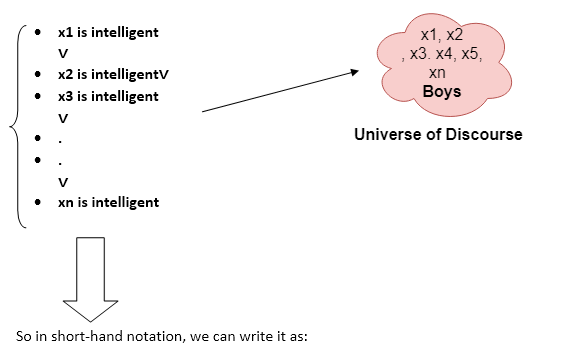
**In Existential quantifier we always use AND or Conjunction symbol (∧).**

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.**



**∃x: boys(x) ∧ intelligent(x)**

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

**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)**.

* 1. **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)**.
  2. **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)**.
  3. **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)].**
  4. **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.**

1. **Knowledge Engineering in First order logic**

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

**The knowledge-engineering process:**

The steps associated with the knowledge engineering process are :

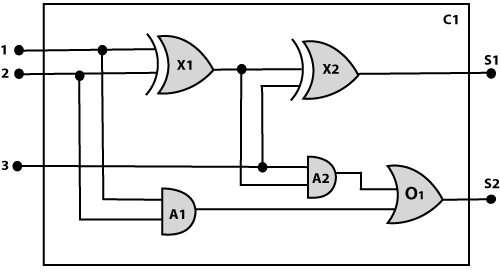
1. **Identify the task:** The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents.
2. **Assemble the relevant knowledge:** The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know-a process called knowledge acquisition.
3. **Decide on a vocabulary of predicates, functions, and constants**: That is, translate the important domain-level concepts into logic-level names. Once the choices have been made. the result is a vocabulary that is known as the ontology of the domain. The word ontology means a particular theory of the nature of being or existence.
4. **Encode general knowledge about the domain:** The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.
5. **Encode a description of the specific problem instance:** For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.
6. **Pose queries to the inference procedure and get answers**: This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing.
7. **Debug the knowledge base:** If an axiom is missing some queries will not be answerable from the knowledge base. Considerable debugging process can solve this.

Example:

x Num Of Legs(x,4) => Mammal(x) Is false for reptiles,amphibians.

**To understand this seven-step process better, we now apply this to the electronic circuit domain**

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



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

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

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

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

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

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

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

1. **Decide on vocabulary**: The next step of the process is to select functions, predicate, and constants to represent the circuits, terminals, signals, and gates. Firstly we will distinguish the gates from each other and from other objects. Each gate is represented as an object which is named by a constant, such as, **Gate()**. The functionality of each gate is determined by its type, which is taken as constants such as **AND, OR, XOR, or NOT**. Circuits will be identified by a predicate: **Circuit ()** .

* For the terminal, we will use predicate: **Terminal(x)**.
* For gate input, we will use the function **In(1, )** for denoting the first input terminal of the gate, and for output terminal we will use **Out (1, )**.
* The function **Arity(c, i, j)** is used to denote that circuit c has i input, j output.
* The connectivity between gates can be represented by predicate

**Connect(Out(1, ), In(1, ))**.

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

1. **Encode general knowledge about the domain:** To encode the general knowledge about the logic circuit, we need some following rules:

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

**∀  :Terminal () ∧ Terminal () ∧ Connect (, ) → Signal () = Signal ()**

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

**∀ t Terminal (t) →Signal (t) = 1 ∨ Signal (t) = 0.**

* Connect predicates are commutative:

**∀   Connect()  ⬄  Connect ().**

* Representation of types of gates:

**∀ g Gate(g) ∧ r = Type(g) → r = OR ∨ r = AND ∨ r = XOR ∨ r = NOT.**

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

**∀  g Gate(g) ∧ Type(g) = AND →Signal (Out(1, g))= 0 ⇔  ∃n Signal (In(n, g))= 0.**

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

**∀  g Gate(g) ∧ Type(g) = OR → Signal (Out(1, g))= 1 ⇔  ∃n Signal (In(n, g))= 1**

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

**∀  g Gate(g) ∧ Type(g) = XOR → Signal (Out(1, g)) = 1 ⇔  Signal (In(1, g)) ≠ Signal (In(2, g)).**

* Output of NOT gate is invert of its input:

**∀  g Gate(g) ∧ Type(g) = NOT →   Signal (In(1, g)) ≠ Signal (Out(1, g)).**

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

**∀  g Gate(g) ∧ Type(g) = NOT →   Arity(g, 1, 1)**

**∀  g Gate(g) ∧ r =Type(g)  ∧ (r= AND ∨r= OR ∨r= XOR) →  Arity (g, 2, 1).**

* All gates are logic circuits:

**∀  g Gate(g) → Circuit (g).**

1. **Encode a description of the problem instance:**

Now we encode problem of circuit C1, firstly we categorize the circuit and its gate components. This step is easy if ontology about the problem is already thought. This step involves the writing simple atomics sentences of instances of concepts, which is known as ontology.

For the given circuit C1, we can encode the problem instance in atomic sentences as below:

Since in the circuit there are two XOR, two AND, and one OR gate so atomic sentences for these gates will be:

1. For XOR gate: Type()= XOR, Type() = XOR
2. For AND gate: Type() = AND, Type()= AND
3. For OR gate: Type () = OR.

And then represent the connections between all the gates.

1. **Pose queries to the inference procedure and get answers:**

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

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

**∃  Signal (In(1**, ) **)=i1  ∧  Signal (In(2, ))=i2  ∧ Signal (In(3, ))= i3**

**∧ Signal (Out(1, )) =0 ∧ Signal (Out(2, ))=1**

1. **Debug the knowledge base:**

Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of knowledge base.

In the knowledge base, we may have omitted assertions like 1 ≠ 0.

**1.2.13 Unification**

* Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
* It takes two literals as input and makes them identical using substitution.
* The UNIFY algorithm takes two sentences and returns a unifier for them if one exists:

Let Ψ1 and Ψ2 be two atomic sentences and 𝜎 be a unifier such that, **Ψ1𝜎 = Ψ2𝜎**, then it can be expressed as **UNIFY(Ψ1, Ψ2)**.

* **Example: Find the MGU for Unify{King(x), King(John)}**
  + Let Ψ1 = King(x), Ψ2 = King(John),
  + **Substitution θ = {John/x}** is a unifier for these atoms and applying this substitution, and both expressions will be identical.
* The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).
* Unification is a key component of all first-order inference algorithms.
* It returns fail if the expressions do not match with each other.
* The substitution variables are called Most General Unifier or MGU.
  + **E.g.** Let's say there are two different expressions, **P(x, y), and P(a, f(z))**.
  + In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

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

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

Following are some basic conditions for unification:

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

**Algorithm: Unify(Ψ1, Ψ2)**

Step. 1: If Ψ1 or Ψ2 is a variable or constant, then:

a) If Ψ1 or Ψ2 are identical, then return NIL.

b) Else if Ψ1is a variable,

a. then if Ψ1 occurs in Ψ2, then return FAILURE

b. Else return { (Ψ2/ Ψ1)}.

c) Else if Ψ2 is a variable,

a. If Ψ2 occurs in Ψ1 then return FAILURE,

b. Else return {( Ψ1/ Ψ2)}.

d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return FAILURE.

Step. 3: IF Ψ1 and Ψ2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ1.

a) Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

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

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

**Implementation of the Algorithm**

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

**Step.2:** Recursively unify atomic sentences:

* 1. Check for Identical expression match.
  2. If one expression is a variable vi, and the other is a term ti which does not contain variable vi, then:
     1. Substitute ti / vi in the existing substitutions
     2. Add ti /vi to the substitution setlist.
     3. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

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

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

            Sol: S0 => Here, Ψ1 = p(f(a), g(Y)), and Ψ2 = p(X, X)  
                  SUBST θ= {f(a) / X}  
                  S1 => Ψ1 = p(f(a), g(Y)), and Ψ2 = p(f(a), f(a))  
                  SUBST θ= {f(a) / g(y)}, **Unification failed**.

Unification is not possible for these expressions.

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

Here, Ψ1 = p(b, X, f(g(Z))) , and Ψ2 = p(Z, f(Y), f(Y))  
S0 => { p(b, X, f(g(Z))); p(Z, f(Y), f(Y))}  
SUBST θ={b/Z}

S1 => { p(b, X, f(g(b))); p(b, f(Y), f(Y))}  
SUBST θ={f(Y) /X}

S2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))}  
SUBST θ= {g(b) /Y}

S2 => { p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))} **Unified Successfully.**

**And Unifier = { b/Z, f(Y) /X , g(b) /Y}.**

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

Here, Ψ1 = {p (X, X), and Ψ2 = p (Z, f(Z))  
S0 => {p (X, X), p (Z, f(Z))}  
SUBST θ= {X/Z}  
              S1 => {p (Z, Z), p (Z, f(Z))}  
SUBST θ= {f(Z) / Z}, **Unification Failed**.

**Hence, unification is not possible for these expressions.**

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

Here, Ψ1 = {prime(11) , and Ψ2 = prime(y)}  
S0 => {prime(11) , prime(y)}  
SUBST θ= {11/y}

S1 => {prime(11) , prime(11)} , **Successfully unified.**  
              **Unifier: {11/y}.**

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

Here, Ψ1 = Q(a, g(x, a), f(y)), and Ψ2 = Q(a, g(f(b), a), x)  
S0 => {Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)}  
SUBST θ= {f(b)/x}  
S1 => {Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))}

SUBST θ= {b/y}  
S1 => {Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))}, **Successfully Unified.**

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

1. **UNIFY(knows(Richard, x), knows(Richard, John))**

Here, Ψ1 = knows(Richard, x), and Ψ2 = knows(Richard, John)  
S0 => { knows(Richard, x); knows(Richard, John)}  
SUBST θ= {John/x}  
S1 => { knows(Richard, John); knows(Richard, John)}, **Successfully Unified.**  
**Unifier: {John/x}.**

**1.2.14 Resolution**

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.

Where **li** and **mj** 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:

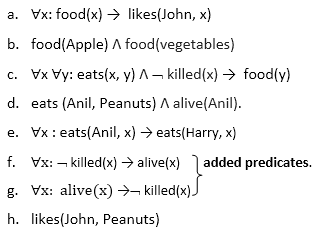
**[Animal (g(x) V ￢ Kills(f(x), x)].**

**Example:**

1. **John likes all kind of food.**
2. **Apple and vegetable are food**
3. **Anything anyone eats and not killed is food.**
4. **Anil eats peanuts and still alive**
5. **Harry eats everything that Anil eats.**  
   **Prove by resolution that:**
6. **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.



**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.

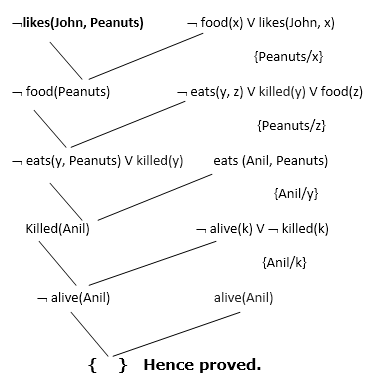
* **Eliminate all implication (→) and rewrite**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
  6. ∀x¬ [¬ killed(x) ] V alive(x)
  7. ∀x ¬ alive(x) V ¬ killed(x)
  8. likes(John, Peanuts).
* **Move negation (¬)inwards and rewrite**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
  6. ∀x ¬killed(x) ] V alive(x)
  7. ∀x ¬ alive(x) V ¬ killed(x)
  8. likes(John, Peanuts).
* **Rename variables or standardize variables**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀w¬ eats(Anil, w) V eats(Harry, w)
  6. ∀g ¬killed(g) ] V alive(g)
  7. ∀k ¬ alive(k) V ¬ killed(k)
  8. likes(John, Peanuts).
* **Eliminate existential instantiation quantifier by elimination.**  
  In this step, we will eliminate existential quantifier ∃, and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.
* **Drop Universal quantifiers.**  
  In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.
  1. ¬ food(x) V likes(John, x)
  2. food(Apple)
  3. food(vegetables)
  4. ¬ eats(y, z) V killed(y) V food(z)
  5. eats (Anil, Peanuts)
  6. alive(Anil)
  7. ¬ eats(Anil, w) V eats(Harry, w)
  8. killed(g) V alive(g)
  9. ¬ alive(k) V ¬ killed(k)
  10. 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.

**Example 2:** **Everyone who loves all animals is loved by someone. Anyone who kills an animal is loved by no one. Jack loves all animals. Either Jack or Curiosity killed the cat, who is named Tuna. Did Curiosity kill the cat?**

Knowledge base:

“Everyone who loves all animals is loved by someone.”

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

“Anyone who kills an animal is loved by no one.”

∀x[∃yAnimal(y) ∧ Kills(x,y)] ⇒ [∀z ¬Loves(z,x)]

“Jack loves all animals”

∀x Animal(x) ⇒ Loves(Jack,x)

“Either Jack or Curiosity killed the cat, who is named Tuna”

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

“Tuna is a cat”

Cat(Tuna)

**1.3 Introduction to logic Programming (PROLOG)**

Prolog is a [logic programming](https://en.wikipedia.org/wiki/Logic_programming) language. It has important role in artificial intelligence. Unlike many other programming languages, Prolog is intended primarily as a declarative programming language. In prolog, logic is expressed as relations (called as Facts and Rules). Core heart of prolog lies at the **logic** being applied. Formulation or Computation is carried out by running a query over these relations.

**Installation in Linux :**

Open a terminal **(Ctrl+Alt+T)** and type:

sudo apt-get install swi-prolog

**Syntax and Basic Fields :**

In prolog, We declare some facts. These facts constitute the Knowledge Base of the system. We can query against the Knowledge Base. We get output as affirmative if our query is already in the knowledge Base or it is implied by Knowledge Base, otherwise we get output as negative. So, Knowledge Base can be considered similar to database, against which we can query. Prolog facts are expressed in definite pattern. Facts contain entities and their relation. Entities are written within the parenthesis separated by comma (, ). Their relation is expressed at the start and outside the parenthesis. Every fact/rule ends with a dot (.). So, a typical prolog fact goes as follows :

Format : relation(entity1, entity2, ....k'th entity).

Example :

friends(raju, mahesh).

singer(sonu).

odd\_number(5).

Explanation :

These facts can be interpreted as :

raju and mahesh are friends.

sonu is a singer.

5 is an odd number.

**Key Features :**  
**1. Unification :** The basic idea is, can the given terms be made to represent the same structure.  
**2. Backtracking :** When a task fails, prolog traces backwards and tries to satisfy previous task.  
**3. Recursion :** Recursion is the basis for any search in program.

**Running queries :**  
A typical prolog query can be asked as :

Query 1 : ?- singer(sonu).

Output : Yes.

Explanation : As our knowledge base contains   
the above fact, so output was 'Yes', otherwise   
it would have been 'No'.

Query 2 : ?- odd\_number(7).

Output : No.

Explanation : As our knowledge base does not   
contain the above fact, so output was 'No'.

* **Some aspects of Prolog fall outside standard logical inference:**
* Prolog uses the database semantics of Section 8.2.8 rather than first-order semantics, and this is apparent in its treatment of equality and negation
* There is a set of built-in functions for arithmetic. Literals using these function symbols are “proved” by executing code rather than doing further inference. For example, the goal “X is 4+3” succeeds with X bound to 7. On the other hand, the goal “5 is X+Y” fails, because the built-in functions do not do arbitrary equation solving.5
* There are built-in predicates that have side effects when executed. These include input– output predicates and the assert/retract predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce confusing results— for example, if facts are asserted in a branch of the proof tree that eventually fails.
* The **occur check** is omitted from Prolog’s unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
* Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes it very fast when given the right set of axioms, but incomplete when given the wrong ones

**Advantages :**  
**1.**Easy to build database. Doesn’t need a lot of programming effort.  
**2.**Pattern matching is easy. Search is recursion based.  
**3.**It has built in list handling. Makes it easier to play with any algorithm involving lists.

**Disadvantages :**  
**1.** LISP (another logic programming language) dominates over prolog with respect to I/O features.  
**2.** Sometimes input and output is not easy.

**Applications :**

Prolog is highly used in artificial intelligence(AI). Prolog is also used for pattern matching over natural language parse trees.

**1.4 Natural Language Processing and Expert system**

Natural Language Processing (NLP) refers to AI method of communicating with an intelligent systems using a natural language such as English.

Processing of Natural Language is required when you want an intelligent system like robot to perform as per your instructions, when you want to hear decision from a dialogue based clinical expert system, etc.

The field of NLP involves making computers to perform useful tasks with the natural languages humans use. The input and output of an NLP system can be −

* Speech
* Written Text

Components of NLP

There are two components of NLP as given −

Natural Language Understanding (NLU)

Understanding involves the following tasks −

* Mapping the given input in natural language into useful representations.
* Analyzing different aspects of the language.

Natural Language Generation (NLG)

It is the process of producing meaningful phrases and sentences in the form of natural language from some internal representation.

It involves −

* **Text planning** − It includes retrieving the relevant content from knowledge base.
* **Sentence planning** − It includes choosing required words, forming meaningful phrases, setting tone of the sentence.
* **Text Realization** − It is mapping sentence plan into sentence structure.

The NLU is harder than NLG.

Difficulties in NLU

NL has an extremely rich form and structure.

It is very ambiguous. There can be different levels of ambiguity −

* **Lexical ambiguity** − It is at very primitive level such as word-level.
* For example, treating the word “board” as noun or verb?
* **Syntax Level ambiguity** − A sentence can be parsed in different ways.
* For example, “He lifted the beetle with red cap.” − Did he use cap to lift the beetle or he lifted a beetle that had red cap?
* **Referential ambiguity** − Referring to something using pronouns. For example, Rima went to Gauri. She said, “I am tired.” − Exactly who is tired?
* One input can mean different meanings.
* Many inputs can mean the same thing.

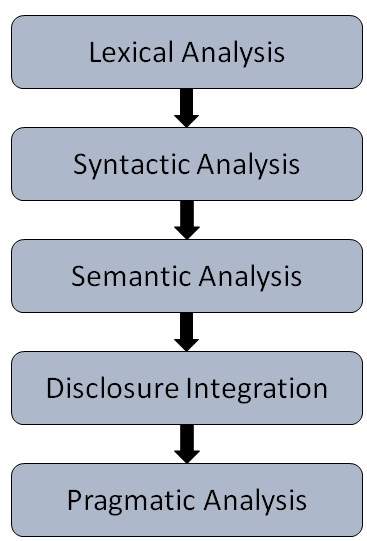
NLP Terminology

* **Phonology** − It is study of organizing sound systematically.
* **Morphology** − It is a study of construction of words from primitive meaningful units.
* **Morpheme** − It is primitive unit of meaning in a language.
* **Syntax** − It refers to arranging words to make a sentence. It also involves determining the structural role of words in the sentence and in phrases.
* **Semantics** − It is concerned with the meaning of words and how to combine words into meaningful phrases and sentences.
* **Pragmatics** − It deals with using and understanding sentences in different situations and how the interpretation of the sentence is affected.
* **Discourse** − It deals with how the immediately preceding sentence can affect the interpretation of the next sentence.
* **World Knowledge** − It includes the general knowledge about the world.

Steps in NLP

There are general five steps −

* **Lexical Analysis** − It involves identifying and analyzing the structure of words. Lexicon of a language means the collection of words and phrases in a language. Lexical analysis is dividing the whole chunk of txt into paragraphs, sentences, and words.
* **Syntactic Analysis (Parsing)** − It involves analysis of words in the sentence for grammar and arranging words in a manner that shows the relationship among the words. The sentence such as “The school goes to boy” is rejected by English syntactic analyzer.



* **Semantic Analysis** − It draws the exact meaning or the dictionary meaning from the text. The text is checked for meaningfulness. It is done by mapping syntactic structures and objects in the task domain. The semantic analyzer disregards sentence such as “hot ice-cream”.
* **Discourse Integration** − The meaning of any sentence depends upon the meaning of the sentence just before it. In addition, it also brings about the meaning of immediately succeeding sentence.
* **Pragmatic Analysis** − During this, what was said is re-interpreted on what it actually meant. It involves deriving those aspects of language which require real world knowledge.

Implementation Aspects of Syntactic Analysis

There are a number of algorithms researchers have developed for syntactic analysis, but we consider only the following simple methods −

* Context-Free Grammar
* Top-Down Parser

Let us see them in detail −

Context-Free Grammar

It is the grammar that consists rules with a single symbol on the left-hand side of the rewrite rules. Let us create grammar to parse a sentence −

“The bird pecks the grains”

**Articles (DET)** − a | an | the

**Nouns** − bird | birds | grain | grains

**Noun Phrase (NP)** − Article + Noun | Article + Adjective + Noun

= DET N | DET ADJ N

**Verbs** − pecks | pecking | pecked

**Verb Phrase (VP)** − NP V | V NP

**Adjectives (ADJ)** − beautiful | small | chirping

The parse tree breaks down the sentence into structured parts so that the computer can easily understand and process it. In order for the parsing algorithm to construct this parse tree, a set of rewrite rules, which describe what tree structures are legal, need to be constructed.

These rules say that a certain symbol may be expanded in the tree by a sequence of other symbols. According to first order logic rule, if there are two strings Noun Phrase (NP) and Verb Phrase (VP), then the string combined by NP followed by VP is a sentence. The rewrite rules for the sentence are as follows −

**S → NP VP**

**NP → DET N | DET ADJ N**

**VP → V NP**

**Lexocon −**

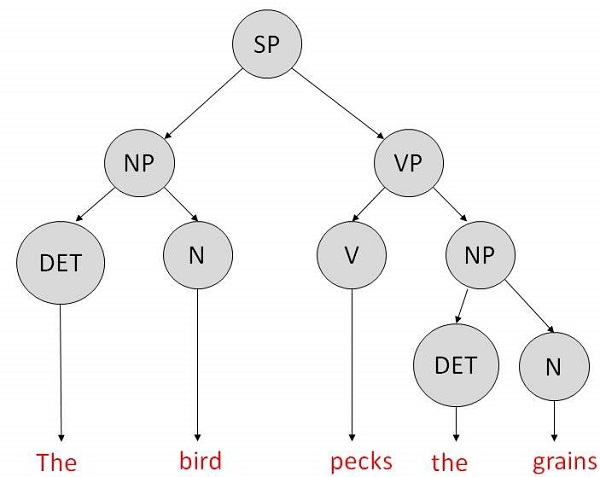
DET → a | the

ADJ → beautiful | perching

N → bird | birds | grain | grains

V → peck | pecks | pecking

The parse tree can be created as shown −



Now consider the above rewrite rules. Since V can be replaced by both, "peck" or "pecks", sentences such as "The bird peck the grains" can be wrongly permitted. i. e. the subject-verb agreement error is approved as correct.

**Merit** − The simplest style of grammar, therefore widely used one.

**Demerits −**

* They are not highly precise. For example, “The grains peck the bird”, is a syntactically correct according to parser, but even if it makes no sense, parser takes it as a correct sentence.
* To bring out high precision, multiple sets of grammar need to be prepared. It may require a completely different sets of rules for parsing singular and plural variations, passive sentences, etc., which can lead to creation of huge set of rules that are unmanageable.

Top-Down Parser

Here, the parser starts with the S symbol and attempts to rewrite it into a sequence of *terminal symbols* that matches the classes of the words in the input sentence until it consists entirely of terminal symbols.

These are then checked with the input sentence to see if it matched. If not, the process is started over again with a different set of rules. This is repeated until a specific rule is found which describes the structure of the sentence.

**Merit** − It is simple to implement.

**Demerits −**

* It is inefficient, as the search process has to be repeated if an error occurs.
* Slow speed of working.

**1.4.1 Expert System**

**Expert systems (ES) are one of the prominent research domains of AI. It is introduced by the researchers at Stanford University, Computer Science Department**.

What are Expert Systems?

The expert systems are the computer applications developed to solve complex problems in a particular domain, at the level of extra-ordinary human intelligence and expertise.

Characteristics of Expert Systems

* High performance
* Understandable
* Reliable
* Highly responsive

Capabilities of Expert Systems

The expert systems are capable of −

* Advising
* Instructing and assisting human in decision making
* Demonstrating
* Deriving a solution
* Diagnosing
* Explaining
* Interpreting input
* Predicting results
* Justifying the conclusion
* Suggesting alternative options to a problem

They are incapable of −

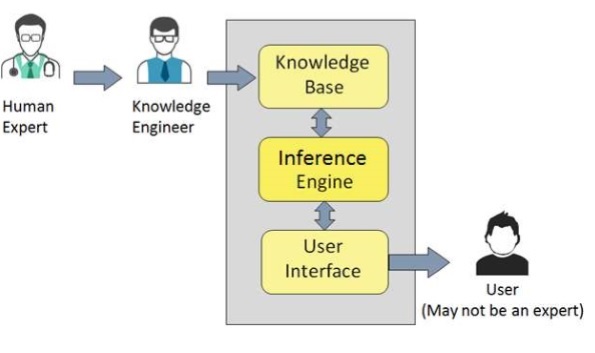
* Substituting human decision makers
* Possessing human capabilities
* Producing accurate output for inadequate knowledge base
* Refining their own knowledge

Components of Expert Systems

The components of ES include −

* Knowledge Base
* Inference Engine
* User Interface

Let us see them one by one briefly −



**Knowledge Base:**

It contains domain-specific and high-quality knowledge.

Knowledge is required to exhibit intelligence. The success of any ES majorly depends upon the collection of highly accurate and precise knowledge.

What is Knowledge?

The data is collection of facts. The information is organized as data and facts about the task domain. **Data, information,** and **past experience** combined together are termed as knowledge.

Components of Knowledge Base

The knowledge base of an ES is a store of both, factual and heuristic knowledge.

* **Factual Knowledge** − It is the information widely accepted by the Knowledge Engineers and scholars in the task domain.
* **Heuristic Knowledge** − It is about practice, accurate judgement, one’s ability of evaluation, and guessing.

**Knowledge representation:**

It is the method used to organize and formalize the knowledge in the knowledge base. It is in the form of IF-THEN-ELSE rules.

**Knowledge Acquisition:**

The success of any expert system majorly depends on the quality, completeness, and accuracy of the information stored in the knowledge base.

The knowledge base is formed by readings from various experts, scholars, and the **Knowledge Engineers**. The knowledge engineer is a person with the qualities of empathy, quick learning, and case analyzing skills.

He acquires information from subject expert by recording, interviewing, and observing him at work, etc. He then categorizes and organizes the information in a meaningful way, in the form of IF-THEN-ELSE rules, to be used by interference machine. The knowledge engineer also monitors the development of the ES.

Inference Engine

Use of efficient procedures and rules by the Inference Engine is essential in deducting a correct, flawless solution.

In case of knowledge-based ES, the Inference Engine acquires and manipulates the knowledge from the knowledge base to arrive at a particular solution.

In case of rule based ES, it −

* Applies rules repeatedly to the facts, which are obtained from earlier rule application.
* Adds new knowledge into the knowledge base if required.
* Resolves rules conflict when multiple rules are applicable to a particular case.

To recommend a solution, the Inference Engine uses the following strategies −

* Forward Chaining
* Backward Chaining

**User Interface:**

User interface provides interaction between user of the ES and the ES itself. It is generally Natural Language Processing so as to be used by the user who is well-versed in the task domain. The user of the ES need not be necessarily an expert in Artificial Intelligence.

It explains how the ES has arrived at a particular recommendation. The explanation may appear in the following forms −

* Natural language displayed on screen.
* Verbal narrations in natural language.
* Listing of rule numbers displayed on the screen.

The user interface makes it easy to trace the credibility of the deductions.

Requirements of Efficient ES User Interface

* It should help users to accomplish their goals in shortest possible way.
* It should be designed to work for user’s existing or desired work practices.
* Its technology should be adaptable to user’s requirements; not the other way round.
* It should make efficient use of user input.

**Expert Systems Limitations**

No technology can offer easy and complete solution. Large systems are costly, require significant development time, and computer resources. ESs have their limitations which include −

* Limitations of the technology
* Difficult knowledge acquisition
* ES are difficult to maintain
* High development costs

Applications of Expert System

The following table shows where ES can be applied.

|  |  |
| --- | --- |
| **Application** | **Description** |
| Design Domain | Camera lens design, automobile design. |
| Medical Domain | Diagnosis Systems to deduce cause of disease from observed data, conduction medical operations on humans. |
| Monitoring Systems | Comparing data continuously with observed system or with prescribed behavior such as leakage monitoring in long petroleum pipeline. |
| Process Control Systems | Controlling a physical process based on monitoring. |
| Knowledge Domain | Finding out faults in vehicles, computers. |
| Finance/Commerce | Detection of possible fraud, suspicious transactions, stock market trading, Airline scheduling, cargo scheduling. |

**Expert System Technology**

There are several levels of ES technologies available. Expert systems technologies include −

* **Expert System Development Environment** − The ES development environment includes hardware and tools. They are −
  + Workstations, minicomputers, mainframes.
  + High level Symbolic Programming Languages such as **LIS**t **P**rogramming (LISP) and **PRO**grammation en **LOG**ique (PROLOG).
  + Large databases.
* **Tools** − They reduce the effort and cost involved in developing an expert system to large extent.
  + Powerful editors and debugging tools with multi-windows.
  + They provide rapid prototyping
  + Have Inbuilt definitions of model, knowledge representation, and inference design.
* **Shells** − A shell is nothing but an expert system without knowledge base. A shell provides the developers with knowledge acquisition, inference engine, user interface, and explanation facility. For example, few shells are given below −
  + Java Expert System Shell (JESS) that provides fully developed Java API for creating an expert system.
  + *Vidwan*, a shell developed at the National Centre for Software Technology, Mumbai in 1993. It enables knowledge encoding in the form of IF-THEN rules.

**Development of Expert Systems: General Steps**

The process of ES development is iterative. Steps in developing the ES include −

* **Identify Problem Domain**
* The problem must be suitable for an expert system to solve it.
* Find the experts in task domain for the ES project.
* Establish cost-effectiveness of the system.
* **Design the System**
* Identify the ES Technology
* Know and establish the degree of integration with the other systems and databases.
* Realize how the concepts can represent the domain knowledge best.
* **Develop the Prototype**

From Knowledge Base: The knowledge engineer works to −

* Acquire domain knowledge from the expert.
* Represent it in the form of If-THEN-ELSE rules.
* **Test and Refine the Prototype**
* The knowledge engineer uses sample cases to test the prototype for any deficiencies in performance.
* End users test the prototypes of the ES.
* **Develop and Complete the ES**
* Test and ensure the interaction of the ES with all elements of its environment, including end users, databases, and other information systems.
* Document the ES project well.
* Train the user to use ES.
* **Maintain the System**
* Keep the knowledge base up-to-date by regular review and update.
* Cater for new interfaces with other information systems, as those systems evolve.
* **Benefits of Expert Systems**
* **Availability** − They are easily available due to mass production of software.
* **Less Production Cost** − Production cost is reasonable. This makes them affordable.
* **Speed** − They offer great speed. They reduce the amount of work an individual puts in.
* **Less Error Rate** − Error rate is low as compared to human errors.
* **Reducing Risk** − They can work in the environment dangerous to humans.
* **Steady response** − They work steadily without getting motional, tensed or fatigued