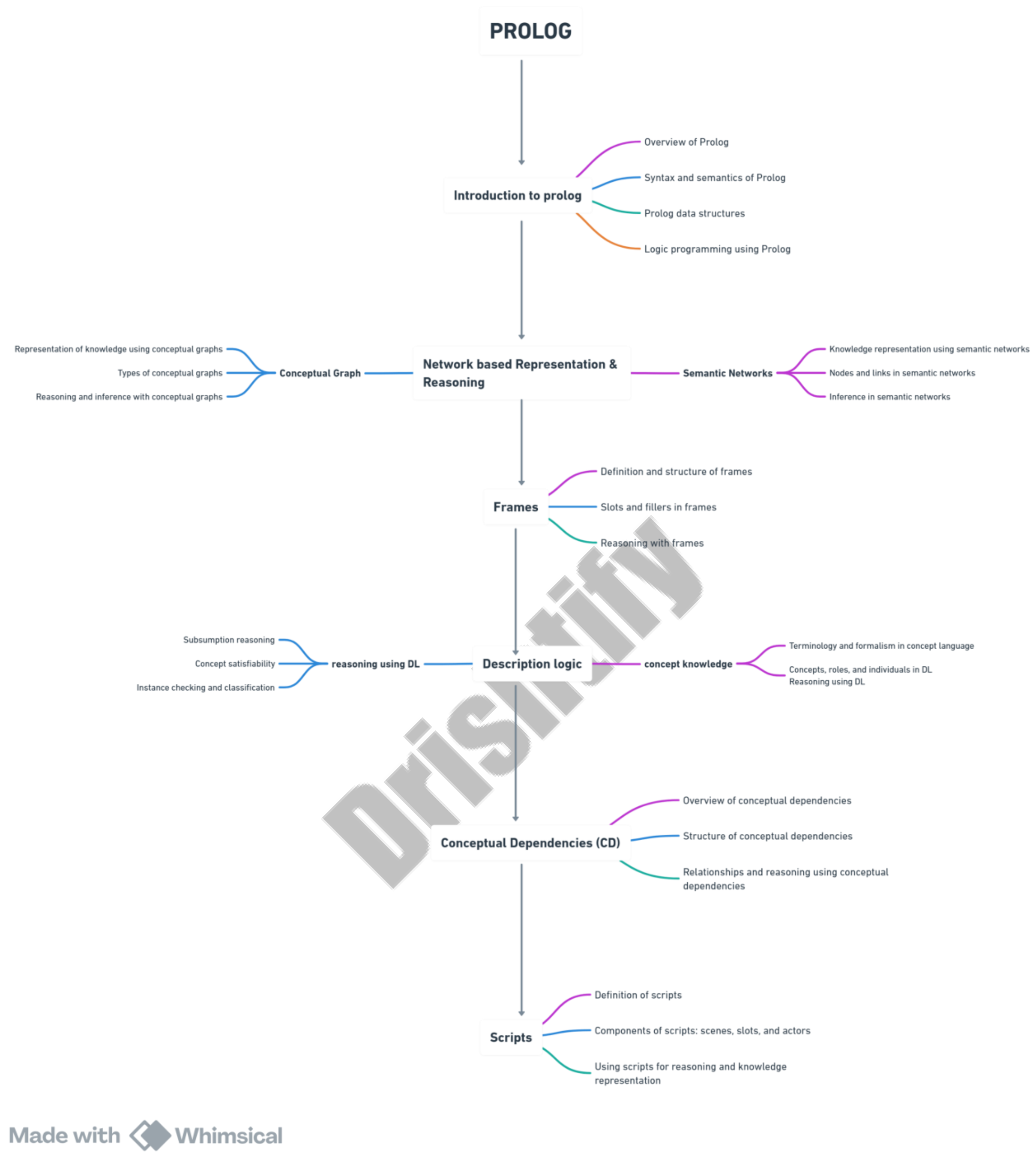
# UNIT-3: PROLOG



PROLOG
Introduction to prolog
Overview of Prolog
Syntax and semantics of Prolog
Prolog data structures
Logic programming using Prolog
Network based Representation & Reasoning
Conceptual Graph
Representation of knowledge using conceptual graphs
Types of conceptual graphs
Reasoning and inference with conceptual graphs
Semantic Networks
Knowledge representation using semantic networks
Nodes and links in semantic networks
Inference in semantic networks
Frames
Definition and structure of frames
Slots and fillers in frames
Reasoning with frames
Description logic
reasoning using DL
Subsumption reasoning
Concept satisfiability
Instance checking and classification
concept knowledge
Terminology and formalism in concept language
Concepts, roles, and individuals in DL Reasoning using DL
Conceptual Dependencies (CD)
Overview of conceptual dependencies
Structure of conceptual dependencies
Relationships and reasoning using conceptual dependencies
Scripts
Definition of scripts
Components of scripts: scenes, slots, and actors
Using scripts for reasoning and knowledge representation
Made with Whimsical

## 1. Introduction to Prolog:

### Overview of prolog:

- Prolog stands for Programming in Logic and is based on the logic programming paradigm. It's a declarative language, meaning the program consists of facts and rules, and you specify what to solve rather than how.

- The user asks a question, and the system searches through facts and rules to find the answer, instead of running a traditional program.
- Marseille Prolog, developed by Colmerauer, was the first Prolog. Later, the Japanese 5th generation project adopted Prolog as a development language in 1981.
- Key features of Prolog include logical variables, backtracking to find proofs, pattern matching, and flexible input/output handling.
- Prolog uses backtracking to explore multiple solutions, which makes it useful for fields like artificial intelligence, including expert systems and natural language processing.
- Prolog is weakly typed, has static scope with dynamic type checking, and shares some similarities with functional languages, especially in using recursive definitions.
- Prolog isn't well-suited for tasks like numerical algorithms or graphics, but it excels in areas involving logic-based reasoning.

### **Applications of prolog:**

- Prolog is widely used in AI for tasks like expert systems, natural language processing, and machine learning.
- It helps build systems that mimic human experts in specific fields by using facts and rules to deduce new information.
- Prolog can parse and understand human languages, making it useful for developing chatbots, translators, and other language-based applications.
- It is used in systems that solve logical problems automatically, such as theorem proving or logic puzzles.
- Prolog works with databases, especially those involving complex queries or hierarchical data structures.
- It’s useful for solving logical puzzles and constraint satisfaction problems like scheduling or route planning.
- Prolog is used in robotics to implement reasoning and decision-making tasks based on sensor data and rules.

## 2. Syntax and Semantics of Prolog:

### 2.1 Syntax:

Prolog's syntax is structured around key components that define its logical constructs

#### a. **Facts:**

Facts are the fundamental assertions in Prolog, representing information about the world.

**General Form:**

```
predicate(argument1, argument2, ...).
```

**Examples:**

```
loves(romeo, juliet).
```

This fact states that "Romeo loves Juliet."

### b. Rules:

Rules define conditional relationships and describe how facts are related. The head of a rule is true if the body is true.

**General Form:**

```
head :- body.
```

**Example:**

```
loves(X, Y) :- loves(X, Z), loves(Z, Y).
```

This means "X loves Y if X loves Z and Z loves Y."

### c. Queries:

Queries are used to ask questions about the knowledge base. They typically start with "?-".

**General Form:**

```
?- predicate(argument1, argument2, ...).
```

**Example:**

```
?- loves(romeo, X).
```

This query asks for whom Romeo loves.

### d. Operators:

Operators in Prolog help in forming complex expressions. Common operators include:

| Operator | Meaning | Example | Explanation |
|---|---|---|---|

| | | | |
|---|---|---|---|
| :- | If | grandparent(X, Y) :- parent(X, Z), parent(Z, Y). | X is a grandparent of Y if X is a parent of Z and Z is a parent of Y. |
| , | And | happy(X) :- rich(X), healthy(X). | X is happy if X is rich and X is healthy. |
| ; | Or | is_animal(X) :- mammal(X); bird(X). | X is an animal if X is a mammal or X is a bird. |

## 2.2 Semantics:

The semantics of Prolog deals with the meaning of the syntactic constructs.

### a. Meaning of Facts and Rules:

- Facts are unconditionally true.
- Rules imply a logical condition that must be met for the head to be true.

### b. Query Interpretation:

When a query is executed, Prolog searches through its database of facts and rules to find matches, employing a method called backtracking to explore possible solutions.

### c. Program structure in prolog:

Prolog programs are composed of facts, rules, and queries. The structure can be summarized as follows:

| Components | Description |
|---|---|
| Predicate | A named relation |
| Clause | A fact or a rule |
| Query | A request for information |

Example:

```
% Facts
parent(john, mary).
parent(mary, sue).
parent(sue, tom).

% Rule (Predicate: grandparent)
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

% Query
?- grandparent(X, tom).
```

Output:

When you run the query " ?- grandparent(X, tom). ", the output will be:

```
X = john ;
X = mary ;
false.
```

Explanation:

- Facts:
  - parent(john, mary). states that John is the parent of Mary.
  - parent(mary, sue). states that Mary is the parent of Sue.
  - parent(sue, tom). states that Sue is the parent of Tom.

- Predicate:
  - grandparent(X, Y) is a predicate that defines the relationship between X (grandparent) and Y (grandchild).It is defined by the rule.

- Query:
  - The query *?- grandparent(X, tom).* asks for all X who are grandparents of Tom. The output shows that both John and Mary are grandparents of Tom.

## 2.3 <u>Variables, Unification, and Resolution in Prolog:</u>

### a. <u>Variables:</u>

In Prolog, variables are placeholders that can represent unknown values. They are identified by capital letters or underscores. When used in queries, Prolog attempts to find specific values that satisfy the given conditions.

## b. Unification

Unification is the process by which Prolog makes two terms identical. It is a key mechanism in Prolog that enables the resolution of queries. During unification, Prolog matches terms (constants, variables, and structures) to determine if they can be made equivalent.

Example:

```
% Facts
parent(john, mary).
parent(mary, sue).

% Query
?- parent(X, mary).
```

Output:

```
X = john ;
false.
```

Explanation:

- In this example, X is a variable that represents the parent of mary.
- Prolog will unify X with john because parent(john, mary) is a matching fact.

## c. Resolution:

Resolution is a method used by Prolog to derive new facts or answer queries. When a query is made, Prolog searches through its knowledge base using unification and backtracking.

Example:

```
% Facts
parent(john, mary).
parent(mary, sue).
parent(sue, tom).

% Rule to define grandparent
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

% Query for grandparent
?- grandparent(X, tom).
```

Output:

```
X = john ;
X = mary ;
false.
```

Explanation:

- **Rule Definition:** grandparent(X, Y) means X is a grandparent of Y if X is a parent of Z and Z is a parent of Y.

- **Query Resolution:**
  - Find Z such that parent(Z, tom) (matches sue).
  - Find X such that parent(X, sue) (matches john and mary).

## 2.4 Common built-in predicates in prolog:

Following are some of the common built-in predicates used in a prolog program in general:

| Predicates | Description | Example | Output | Meaning |
|---|---|---|---|---|
| **is** | Evaluates an arithmetic expression | X is 2+3. | X=5. | Assigns 5 to X. |
| **=** | Checks if two terms can be unified. | X=5. | X=5. | Unifies X with 5. |
| **\=** | Checks if two terms cannot be unified. | x\=5 | true/false | True if X is not 5. |
| **member** | Checks if an element is a member of a list. | member(X,[1,2,3]). | true/false | True if X is 1,2, or 3. |

## 2.5 Control structures in prolog:

| Control structure | description | Example | output |
|---|---|---|---|
| Conjunction(,) | Both conditions must be true(logical AND) | ?-parent(john,mary), parent(mary,sue). | True. |
| Disjunction(;) | Atleast one of the conditions must be true(Logical OR) | ?-parent(john,mary); parent(sue,tom). | True.(if either condition holds) |
| Negation(\+) | Succeeds if the condition cannot be proven true(Logical NOT) | ?-\+parent(john,tom) | True.(if john is not tom's parent) |
| If-then(->) | Executes the "then" part if | ?-parent(john,mary)->write('yes'). | Yes(if john is mary's parent) |

| | | | |
|---|---|---|---|
| | the “if” part is true | | |
| If-then-else(-> ;) | Executes the “then” part of the “if” part is true otherwise the “else” part. | ?-parent(john,sue)->write(‘yes’); write(‘no’). | No(as john is not sue’s parent) |

## 2.6 Input/Output operations in prolog:

| Operations | Description | Example | Output |
|---|---|---|---|
| write | Outputs term to console. | ?- write('Hello, World!'). | Hello, World! true. |
| nl | Prints a new line | ?- write('Hello,'), nl, write('World!'). | Hello, World! |
| read | Reads user input and unifies it with a variable | ?- read(X). (input: apple) | X = apple. |
| tab | Prints specified number of spaces | ?- write('Hello'), tab(4), write('World'). | Hello World |
| put | Outputs the ASCII character corresponding to the integer | ?- put(65). | A |
| get | Reads the character from the input and returns its ASCII value. | ?- get(X). (input: A) | X = 65. |

## 3. Prolog data structures:

Prolog provides several data structures to represent and manipulate information. These include atoms, numbers, variables, lists, and compound terms. Let's dive into each of these with examples to make things simpler.

- **Atom**: An atom is a constant that represents a specific value. Atoms in Prolog are usually written as lowercase words or enclosed in single quotes.

  **Example:**
  color(red).
  color('Blue sky').

  Query:
  ?- color(red).

  Output:
  True

- **Numbers**: Prolog supports integers and floating-point numbers. These are written just like in other programming languages.

  **Example:**
  age(25).
  height(5.9).

  Query:
  ?- age(25).

  Output:
  true

- **Variables**: Variables in Prolog are placeholders and must start with an uppercase letter or an underscore. Prolog uses them to find possible matches.

  **Example:**
  person(Name, Age).

  Query:
  ?- person(X, 25).
  Output:
  X = some_name  % Prolog would try to find a person with age 25

- **Lists**: Lists are one of the most important data structures in Prolog. A list is a sequence of items enclosed in square brackets ([]). The first element is called the head, and the rest is called the tail.

**Example:**

```
fruits([apple, banana, cherry]).
```

Query:

```
?- fruits([X|Y]).
```

Output:

```
X = apple,
Y = [banana, cherry]
```

**Misc. example:**

```
Fruits([apple | [banana, cherry]]).
```

Query:

```
?- fruits([apple | [banana, cherry]]).
```

Output:

```
true
```

- **Compound Terms:** Compound terms represent structured data. They have a functor (name) and arguments.

  **Example:**

  ```
  point(X, Y).
  ```

  Query:

  ```
  ?- point(3, 4).
  ```

  Output:

  ```
  true
  ```

# 4. Programming logic in prolog:

## Structure of prolog program:

- **Facts Section**: Contains basic facts.
  Example:
  ```
  cat(whiskers). dog(buddy).
  ```

- **Rules Section**: Contains rules for inference.
  Example:

pet(X) :- cat(X). pet(X) :- dog(X).

Facts and Rules being written together as:

```
% Facts
cat(whiskers).
dog(buddy).
% Rules
pet(X) :- cat(X).
pet(X) :- dog(X).
```

- **Queries Section**: Asks questions.
  Example:
  ?- pet(buddy).

- **Output section:**
  true.

## 5. Network based representation in prolog:

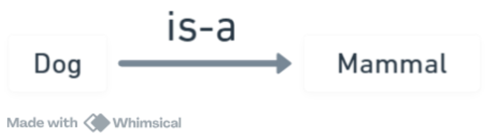
Network-based representation is a way to represent knowledge using nodes and links that capture relationships between entities. Two primary types of network-based representations are **Semantic Networks** and **Conceptual Graphs**. These structures help to model information in a structured way that allows for reasoning, inference, and querying knowledge.

### a. Semantic Networks:

A **Semantic Network** is a graphical structure used to represent knowledge in the form of concepts (nodes) and relationships (links). It is a type of knowledge representation that depicts information using graph-like diagrams, making it easier to model associations between concepts.

- **Knowledge Representation Using Semantic Networks:**
  Semantic networks represent knowledge as interconnected nodes. Each **node** represents a concept or entity, and each **link** (edge) defines a relationship between two nodes. These relationships can indicate attributes, hierarchical information (like parent-child), or causal relations.

  **Example**: If we want to represent that "A dog is a mammal," we can create a node for **Dog** and another for **Mammal**, linking them with the "is a" relationship.



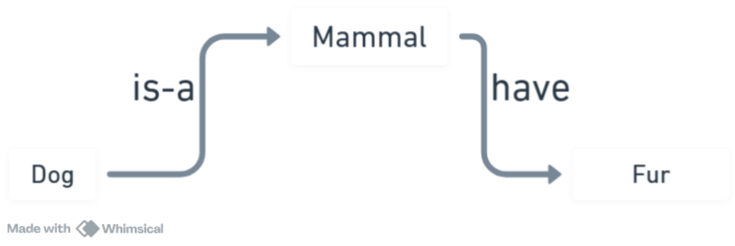
- **Components of semantic network:**
  Semantic Network has several key components:

  - **Nodes**: Represent concepts or entities. Each node can carry additional information, like attributes or properties.
  - **Links (Edges):** Represent relationships between nodes. Links can have types (e.g., "is a," "has a," "part of") to describe the nature of the relationship.
  - **Labels**: Links can have labels to specify the type of relationship, making the network more informative.
  - **Weights (optional):** Some semantic networks may include weights on links to signify the strength or importance of the relationship.
  - **Hierarchy**: Semantic networks can also represent hierarchical relationships, showing parent-child relationships among nodes.

- **Inference in semantic networks:**
  Inference in semantic networks involves deriving new knowledge based on existing relationships and nodes. This can be done through:

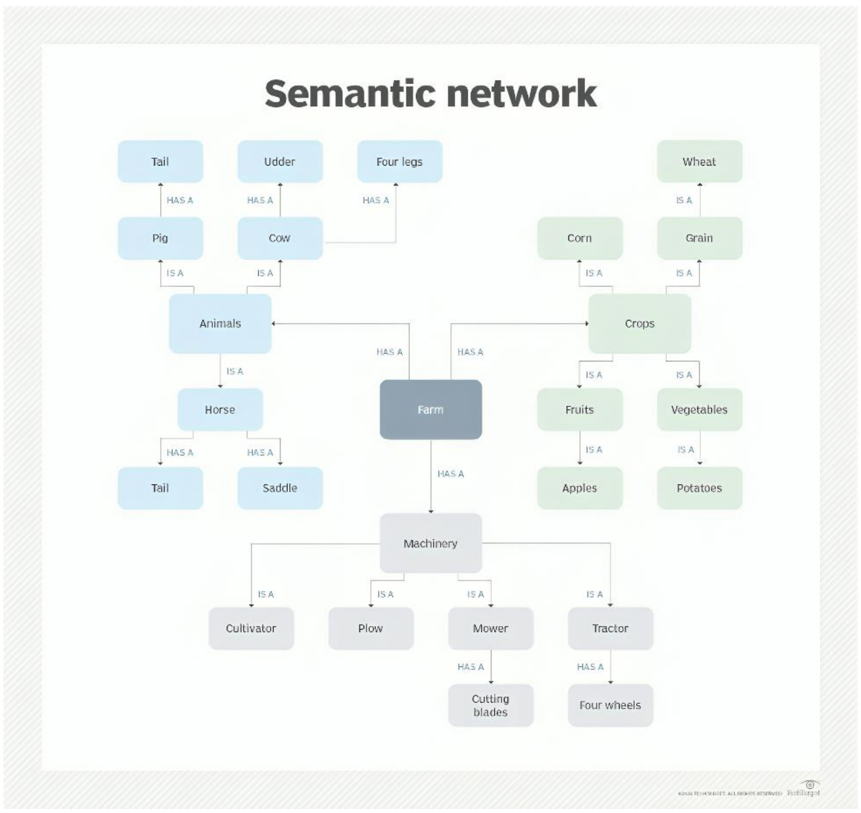
  - Path Traversal:By following links between nodes, we can infer new relationships.
  - Chain Rules: If we know that "A is a B" and "B has a property C," we can infer that "A has property C."

  **Example of Inference:**



Hence, from this we can infer “Dog has Fur”.

Example of Semantic Network:



This semantic network can be inferred as:

- The horse has a tail.
- A cow has four legs.
- Animals like pigs and cows are found on farms.
- The tractor has four wheels.
- A pig is a type of animal.
- Fruits like apples and vegetables like potatoes are types of crops.
- Tom is owned by Rashan.
- Tom is brown in color.
- Dogs like bones.
- The dog sat on the mat.
- A dog is a mammal.
- A cat is an instance of an animal.
- All mammals are animals.
- Mammals have fur.

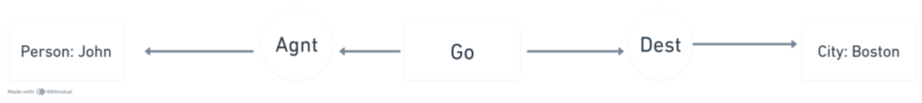
## b. Conceptual graphs:

A **Conceptual Graph (CG)** is a more formal method of representing knowledge than semantic networks. Conceptual graphs are graphs that represent propositions using nodes for **concepts** and **conceptual relations**. They capture both entities and their relationships in a format that can be used for reasoning.

➢ **Knowledge Representation Using Conceptual Graph:**

Conceptual graphs represent **entities** (concepts) and their relationships through **relations**. Each node represents a concept, and relationships between nodes are represented using edges.

**Example**: John is going to Boston



In Display Form(DF), this figure shows a conceptual graph with three concepts: [Go], [Person: John] and [City: Boston]. It has three conceptual relations: (Agnt) relates [Go] to the agent John, (Dest) relates [Go] to the destination Boston.

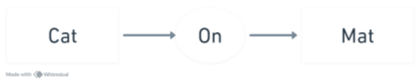
Since the concept [Go] is attached to two conceptual relations, the linear form cannot be drawn in a straight line, as in the figure. Instead, a hyphen at the end of the first line indicates that the relations attached to [Go] are continued on subsequent lines.

In Linear Form(LF):

```
[Go]-
  (Agnt)->[Person: John]
  (Dest)->[City: Boston]
```

- **Types of Conceptual Graphs:**
  Conceptual graphs can vary depending on how concepts and relations are structured:

  - **Simple Conceptual Graphs:** Basic graphs with simple relations. Example: A cat is on a mat.

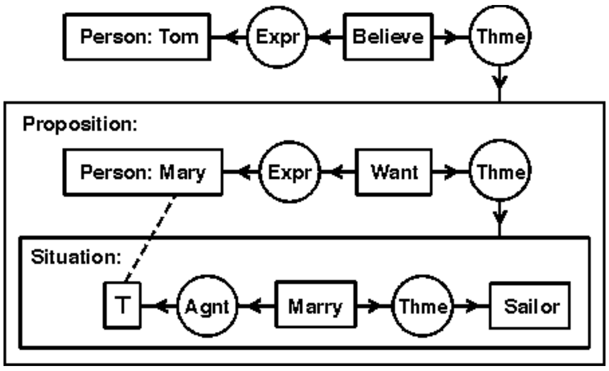
    In the display form (DF), concepts are shown as rectangles (e.g., [Cat], [Mat]), and conceptual relations are represented by circles/ovals (e.g., (On)). Arrows link relations to concepts: the first arrow points toward the relation, and the second points away. If a relation has more than two arcs, they are numbered.

    In the linear form (LF), concepts are represented by square brackets instead of boxes, and the conceptual relations are represented by parentheses instead of circles:

[Cat]->(On)->[Mat].

- **Nested Conceptual Graphs:** Contain sub-graphs inside nodes for complex representations.

  **Example:** Tom believes that Mary wants to marry a sailor.



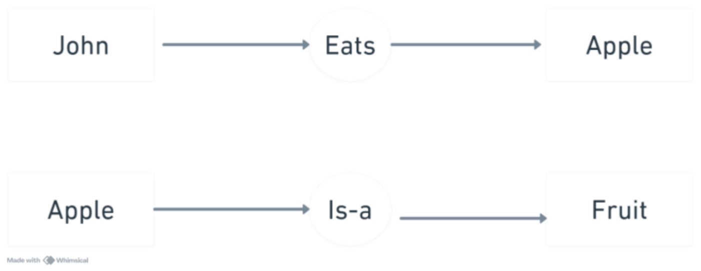
  In Display Form(DF), In Figure 4, the concept of type Proposition represents what Tom believes, and inside it is a Situation that describes what Mary wants. The graph represents the sentence: "Tom believes that Mary wants to marry a sailor."

  In Figure 5, Tom is the experiencer of [Believe], linked by the theme relation (Thme) to a proposition. Inside the proposition, Mary is the experiencer of [Want], with a situation as the theme, where Mary (represented by [T]) marries a sailor. A coreference link shows [T] refers to the same [Person: Mary].

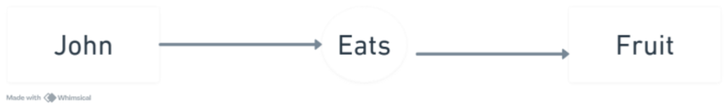
  In Linear Form(LF),
  [Person: Tom]<-(Expr)<-[Believe]->(Thme)-
    [Proposition: [Person: Mary *x]<-(Expr)<-[Want]->(Thme)-
      [Situation: [?x]<-(Agnt)<-[Marry]->(Thme)->[Sailor]]].

➢ **Reasoning and inference with conceptual graphs:**
**Reasoning and inference in conceptual graphs** derive new knowledge by combining concepts and relations. Using operations like **projection**, graphs match and infer new facts. For example, from "John eats an apple" and "Apple is a fruit," we infer "John eats a fruit." This enables logical conclusions.

**Example**:



By combining these facts we can infer that:



## 6. Frames:

### a. Definition of Frames:

A frame is a data structure used to represent knowledge in artificial intelligence (AI) systems. It organizes information about an object, event, or situation by grouping related facts into one structure. A frame is similar to a record in a database, where fields (or slots) store various details about the object being described.

Frames are used in knowledge representation to help machines understand and reason about the world, making decisions based on the knowledge stored in these structured forms.

### b. Structure of frames:

Frames consist of:

- **Frame Name:** The name of the object or concept being described.
- **Slots**: The attributes or properties of the object.
- **Fillers**: The values assigned to the slots.

**Example:**

| **Frame (Dog)** | |
|---|---|
| **Slots** | **Fillers** |
| **Name** | **Ivy** |
| **Age** | **7yrs.** |
| **Gender** | **Female** |
| **Owner** | **Drishti** |

In this example, "Dog" is the frame name, and it has slots like "Name", "Age", "Gender" and "Owner." Each slot is filled with relevant information, like "Ivy" for Name, "7yrs." for age, "Female" for Gender and "Drishti" for Owner.

**Slots and Fillers in Frames:**

- **Slots** are placeholders for various properties of the object. They describe different aspects or attributes.
- **Fillers** are the specific values or data that are assigned to the slots. Fillers can be simple data (like numbers or strings), or they can be more complex structures (like another frame or a set of rules).

## c. <u>Reasoning with Frames:</u>

Consider 2 Frames:

| **Frame 1 (Animal)** | |
|---|---|
| **Slots** | **Fillers** |
| **Type** | **Mammal** |
| **Color** | **Brown** |
| **Habitat** | **Forest** |
| **Diet** | **Herbivore** |

| **Frame 2 (Animal1)** | |
|---|---|
| **Slots** | **Fillers** |
| **Type** | **Deer** |
| **Color** | **Brown** |
| **Habitat** | **Forest** |
| **Diet** | **Herbivore** |

Reasoning:

- Is Animal1 a mammal?
  **Conclusion**: Yes (Type: Mammal).
- Where does Animal1 live?
  **Conclusion**: Forest (Habitat: Forest).

- What does Animal1 eat?
  **Conclusion**: Plants (Diet: Herbivore).

## 7. Description Logic:

### a. Concept Knowledge:

➢ **Definition:**
Concept knowledge in DL refers to the way knowledge about classes of objects (concepts), their properties (roles), and specific instances (individuals) is represented. It allows us to formalize and reason about the relationships and characteristics of various entities within a domain.

➢ **Components of Concept Knowledge:**
Terminology and Formalism in Concept Language:

- **Concepts**: Represents classes or sets of objects. For example, Person can represent all people.
- **Roles**: Represents binary relationships between concepts. For instance, hasChild denotes a relationship between a Parent and a Child.
- **Individuals**: Specific instances of concepts. For example, Alice and Bob can be instances of the Person concept.

➢ **Formalism:**
It defines the syntax for representing concepts (uppercase letters), roles (uppercase letters), and individuals (lowercase letters) to facilitate knowledge representation and reasoning:

- **Concepts**: Denoted by uppercase letters (e.g., C,D).
- **Roles**: Denoted by uppercase letters (e.g., R,S).
- **Individuals**: Denoted by lowercase letters (e.g., a,b).

➢ **Operators:**

| Operator | Symbol | Example | Description |
|---|---|---|---|
| Union | **C ∪ D** | **Person ∪ Employee** | **Combines Person and Employee** |
| Intersection | **C ∩ D** | **Parent ∩ Adult** | **Represents Parent that is also Adult** |
| Complement | **¬ C** | **¬ Person** | **Includes nonperson individuals** |
| Existential Quantification | ∃R.C | ∃hasChild.Person | **Denotes individuals with hasChild relationship with Person** |
| Universal Quantification | ∀R.C | ∀hasChild.Person | **Denotes all individuals with hasChild relationship are Person** |

- **Example:**
  Parent=Person∩∃hasChild.Person

  This definition states that a Parent is a Person who has at least one Child.

## b. <u>Reasoning Using DL</u>

- **Definition:**
  Reasoning using DL refers to the processes by which new knowledge is inferred from the existing knowledge represented in the DL framework. This includes checking the relationships between concepts, determining satisfiability, and verifying instances.

- **Key reasoning tasks:**
  1. **Subsumption reasoning:**
     Determines if one concept is more general than another. If concept A is subsumed by concept B (denoted A⊆B), every instance of A is also an instance of B.

     **Example**:

Let A be Mother and B be Parent. The relationship Mother⊆Parent holds, indicating every Mother is a Parent.

| Concept | Subsumed by |
|---|---|
| Mother | Parent |
| Child | Person |

2. **Concept satisfiability:**
   Checks whether a concept can have any instances. A concept is satisfiable if it is possible for it to have individuals that belong to it.

   **Example**:
   Consider the concept C=Parent∩Child. If we can find an individual who satisfies both conditions,C is satisfiable.

   **Output:**
   If Parent(Alice) and Child(Alice) exist, then C is satisfiable.

3. **Instance checking and Classification:**

   - **Instance checking:** Verifies if a specific individual belongs to a concept.

     **Example**:
     To check if Alice is a Parent, we verify the fact Parent(Alice).

     **Output**:
     - **Input**: Parent(Alice)
     - **Output**: true (if Alice is indeed a parent)

   - **Classification:** Organizes concepts into a hierarchy based on their relationships. Classification identifies the subsumption relationships among concepts.

     **Example**:
     Consider the hierarchy:

     Person
     - Parent
       - Mother
       - Father
     - Child

This hierarchy helps in understanding the relationships between various concepts and supports efficient reasoning.

## c. Conceptual Dependencies (CD):

- **Overview of Conceptual Dependencies:**

  Conceptual Dependencies (CD) is a semantic model developed by Roger Schank in the 1970s to represent the meanings of natural language sentences. Its primary objective is to capture the underlying semantics of actions and events, allowing for a deeper understanding beyond mere linguistic forms. By representing the meanings of sentences through a structured framework, CD aids in natural language processing, enabling machines to interpret and reason about human language.

  Key features of Conceptual Dependencies include:

  - **Semantics over Syntax:** CD emphasizes meaning rather than the specific words used to express it.
  - **Primitive Concepts:** It utilizes a set of predefined primitive actions (e.g., GIVE, GET, SEE) to represent various actions and their participants.
  - **Reasoning Capability:** The structure facilitates reasoning about relationships and implications inherent in the represented actions.

- **Structure of conceptual dependencies:**
  The structure of CD revolves around several fundamental components:

  1. **Primitive Acts (Primitives):**
     Primitive acts are the foundational structural blocks of all activities in CD. They represent central activities that can occur in various contexts and are language-independent. Common primitive acts include:

     - ATRANS: The exchange of a theoretical relationship (e.g., giving data).
     - PTRANS: The actual movement of an object from one place to another.
     - PROPEL: The application of physical force to move an object.
     - MOVE: A self-motivated change in position by an animate object.
     - INGEST: Taking something into the body (e.g., eating).
     - EXPEL: Forcing something out of the body (e.g., exhaling).

- SPEAK: Producing verbal output.
- ATTEND: Directing sensory organs towards a stimulus (e.g., looking).

2. **Conceptual Cases (Cases):**
   Conceptual cases describe the roles played by various entities in an action, helping to clarify who is doing what to whom. Common cases include:

   - Agent (AG): The entity performing the action.
   - Object (OB): The entity affected by the action.
   - Recipient (RE): The entity receiving the result of the action.
   - Instrument (IN): The means used to perform the action.
   - Source (SRC): The starting point of a transfer action.
   - Destination (DEST): The endpoint of a transfer action.
   - Experiencer (EX): The entity experiencing a sensation or feeling.

3. **Modifiers:**
   Modifiers provide additional details about actions, objects, or other components in the CD structure. They specify characteristics such as:

   - Time: When the action occurs (e.g., yesterday, now).
   - Location: Where the action occurs (e.g., in the park, at home).
   - Manner: How the action is performed (e.g., quickly, carefully).
   - Reason: Why the action is performed (e.g., to earn money, for fun).

4. **Conceptual Tenses:**
   Conceptual tenses convey the temporal aspects of actions, aiding in understanding when actions happen and their duration. Examples include:

   - Past: Actions that have already happened.
   - Present: Actions currently happening.
   - Future: Actions that will happen.
   - Progressive: Ongoing actions.
   - Perfect: Completed actions.

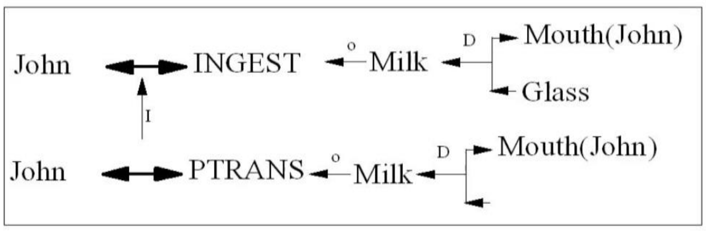
5. **Dependencies:**
   These are relationships between actions that indicate how they are connected. Examples include:

   - Causal Dependency: One action causes another.
   - Temporal Dependency: The sequence of actions in time.
   - Conditional Dependency: One action depends on the occurrence of another.

6. **State Descriptions:**
   State descriptions depict the condition of entities during actions, including physical (e.g., location, ownership) or mental states (e.g., beliefs, desires).

➢ **Relationships and Reasoning Using Conceptual Dependencies:**

• John drank milk.



**Explanation:**

- **PTRANS** (Physical Transfer): John moves Milk from Glass to Mouth(John) (John physically transfers the milk to his mouth).
- **INGEST** (Ingesting Action): John takes the Milk from Mouth(John) (John consumes the milk after it's in his mouth).
- **Key Symbols:**
  - ➢ D (Direction): Shows the flow of the action.
  - ➢ o(Object): Refers to the object (Milk) involved in the action.
  - ➢ I (Ingest): Represents the act of consuming or ingesting the object.

## d. Scripts:

➢ **Definition of scripts:**
Scripts are structured frameworks used in artificial intelligence to represent stereotypical sequences of events, capturing everyday activities in an organized manner. They consist of key components that allow AI

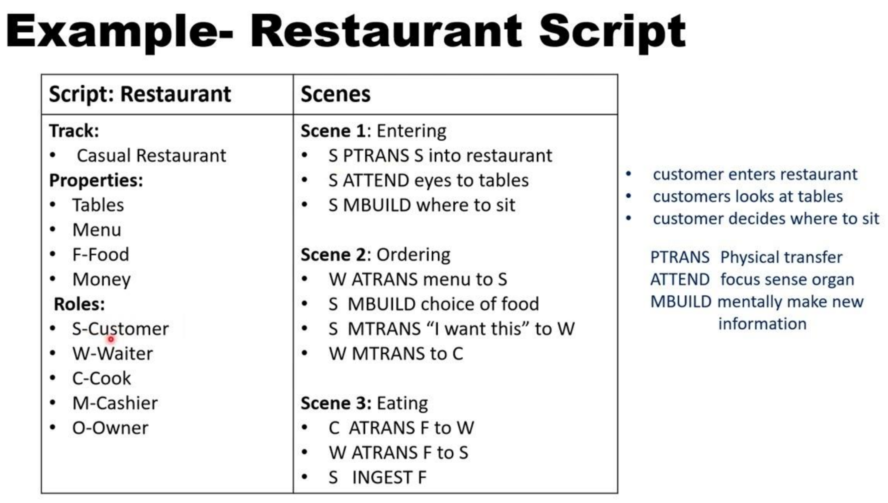
systems to predict, infer, and interpret actions in a logical sequence, providing context and understanding.

➢ **Components of scripts:**

Scripts are composed of several key components:

- Scenes: The basic units of a script, detailing specific actions or events.
- Actors: The entities (e.g., people, robots) performing actions within the scenes.
- Props: Objects involved in the actions.
- Entry Conditions: Preconditions that must be met for a script to initiate.
- Results: The outcomes or goals achieved by completing the script.

➢ **Using Scripts for knowledge representation:**



**Example- Restaurant Script**

| **Script: Restaurant** | **Scenes** |
|---|---|
| **Track:** | **Scene 1**: Entering |
| • Casual Restaurant | • S PTRANS S into restaurant |
| **Properties:** | • S ATTEND eyes to tables |
| • Tables | • S MBUILD where to sit |
| • Menu | |
| • F-Food | **Scene 2**: Ordering |
| • Money | • W ATRANS menu to S |
| **Roles:** | • S MBUILD choice of food |
| • S-Customer | • S MTRANS "I want this" to W |
| • W-Waiter | • W MTRANS to C |
| • C-Cook | |
| • M-Cashier | **Scene 3:** Eating |
| • O-Owner | • C ATRANS F to W |
| | • W ATRANS F to S |
| | • S INGEST F |

- customer enters restaurant
- customers looks at tables
- customer decides where to sit

| | |
|---|---|
| PTRANS | Physical transfer |
| ATTEND | focus sense organ |
| MBUILD | mentally make new information |

***MTRANS**: Mental transfer of information, like knowledge or thoughts, between entities.
***ATRANS**: Physical transfer of objects or possessions from one entity to another.
***INGEST**: The action of taking in food or substances into the body.

===========X=========X==========X===========X=========