

E-Contents

Program : B.tech AI 4th Semester

Subject Code: EAI404

Subject Name: Artificial Intelligence (AI)

Unit-1

1. Introduction to Artificial Intelligence

Artificial Intelligence (AI) is the simulation of human intelligence processes by machines, especially computer systems. These processes include learning (acquiring information and rules for using it), reasoning (using rules to reach approximate or definite conclusions), and self-correction.

- Key Points:
- AI enables machines to mimic human behavior.
- It powers technologies like virtual assistants, self-driving cars, recommendation systems, etc.

2. History of Artificial Intelligence

- 1950s: Alan Turing introduces the concept of a machine that can simulate any human intelligence task.
- 1956: The term “Artificial Intelligence” is coined by John McCarthy at the Dartmouth Conference.
- 1960s–70s: Early AI programs developed (e.g., ELIZA, SHRDLU).
- 1980s: Rise of expert systems.
- 1997: IBM's Deep Blue defeats world chess champion Garry Kasparov.
- 2010s: Deep learning and big data revolutionize AI (e.g., Siri, Alexa, self-driving cars).
- Today: AI is integrated into many aspects of daily life, with rapid advances in deep learning and natural language processing.

3. The Turing Test

Proposed by Alan Turing in 1950, the Turing Test evaluates a machine’s ability to exhibit intelligent behavior equivalent to or indistinguishable from that of a human.

- Key Concept:
- If a human cannot distinguish the machine from a human in conversation, the machine is said to have passed the Turing Test.

4. Founding Blocks and Subfields of AI

- Foundational Elements:
 - Knowledge Representation

- Reasoning
- Learning
- Perception
- Natural Language Processing
- Major Subfields:
 - Machine Learning (ML)
 - Natural Language Processing (NLP)
 - Robotics
 - Computer Vision
 - Expert Systems
 - Neural Networks

5. Application Areas of AI

- Healthcare (diagnostics, personalized medicine)
- Finance (fraud detection, trading bots)
- Education (smart tutoring systems)
- Transportation (autonomous vehicles)
- Retail (recommendation systems)
- Cybersecurity (threat detection)

6. Applications of Artificial Intelligence

- Expert System
 - Mimics decision-making abilities of a human expert. Uses a knowledge base and inference rules. Example: MYCIN (medical diagnosis).
- Intelligent Agents
 - Autonomous entities that perceive the environment and act to achieve goals. Used in software (chatbots, recommendation agents) and robotics.
- Computer Vision
 - Enables machines to interpret and make decisions based on visual data. Applications: facial recognition, autonomous vehicles, medical image analysis.
- Natural Language Processing (NLP)
 - Allows machines to understand, interpret, and generate human language. Applications: machine translation, sentiment analysis, chatbots.

1. State Space Search

- A problem-solving approach where all possible states of a system are represented.
- Each state represents a snapshot of the system.
- Actions transform one state to another.
- Search algorithms explore these states to find a solution path from the **initial state** to a **goal state**.

2. Production System

- A model of computation with:
 - **A set of rules** (productions)
 - **A working memory** (current state)
 - **A control strategy** (rule selection mechanism)
- Used in AI systems to simulate human reasoning.

🔍 Problem Characteristics

- **Well-defined vs ill-defined:** Goals and solutions are either clearly stated or vague.
- **Static vs dynamic:** Whether the problem changes during solution finding.
- **Deterministic vs stochastic:** Whether actions lead to predictable outcomes.
- **Single-agent vs multi-agent:** Number of entities acting in the environment.
- **Observable vs partially observable:** Access to complete or partial information.
- **Discrete vs continuous:** Finite set of possibilities or infinite.

DFS (Depth-First Search)

- Explores as deep as possible along each branch before backtracking.
- **Uses a stack (LIFO)** (can be implemented via recursion).
- **Pros:**
 - Memory-efficient.
 - Good for problems with deep solutions.
- **Cons:**
 - Can get stuck in infinite paths.
 - Not guaranteed to find the shortest path.

DFS Pseudocode (Recursive):

```
python
CopyEdit
def DFS(node):
    if node is goal:
        return True
    for child in node.children:
        if DFS(child):
            return True
    return False
```

BFS (Breadth-First Search)

- Explores all neighbors before going to the next level.
- **Uses a queue (FIFO).**
- **Pros:**
 - Finds shortest path (in unweighted graphs).
- **Cons:**
 - Uses a lot of memory (stores all nodes at current level).

BFS Pseudocode:

```
python
CopyEdit
def BFS(start):
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node is goal:
            return True
        queue.extend(node.children)
```

⊗ Limitations of DFS and BFS

Algorithm	Limitations
DFS	May not find the shortest path, can go infinitely deep
BFS	High memory usage, slow with large depth

🔍 Heuristic Search Techniques

- Use **heuristics**: educated guesses or rules of thumb to make better choices.
 - Improve efficiency by prioritizing paths likely to lead to a solution.
-

Generate and Test

- Naive approach.
 - Generate possible solutions and test them until the correct one is found.
 - Simple, but inefficient for large problems.
-

Hill Climbing

- A local search algorithm.
- Moves in the direction of increasing value (higher heuristic score).

- Stops at local maxima.

Types:

- **Simple hill climbing**
- **Steepest ascent hill climbing**
- **Stochastic hill climbing**

Limitation: Can get stuck at local maxima, ridges, or plateaus.

Best-First Search

- Uses an evaluation function to decide which node to expand next.
 - Often implemented with a **priority queue**.
 - Greedy approach: selects node that looks closest to goal.
-

Problem Reduction

- Breaks a complex problem into smaller subproblems.
 - Solve each subproblem, then combine solutions.
-

Constraint Satisfaction

- Problems defined by:
 - **Variables**
 - **Domains**
 - **Constraints**
- Goal: Assign values to variables satisfying all constraints.

Examples:

- Sudoku
 - Map coloring
 - Scheduling
-

Cryptarithmic Problems

- Type of constraint satisfaction problem.
- Letters represent digits.
- Find digit assignments so that arithmetic equations hold.

Example:

```

markdown
CopyEdit
  SEND
+ MORE
-----
  MONEY

```

Unit III

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts.

- The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:
 - Forward chaining
 - Backward chaining

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm.

- Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the first-order definite clause.

Definite clause: A clause which is a disjunction of literals with exactly one positive literal is known as a definite clause or strict horn clause.

- **Horn clause:** A clause which is a disjunction of literals with at most one positive literal is known as horn clause. Hence all the definite clauses are horn clauses.

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

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

- Forward chaining is also known as a forward deduction or forward reasoning method when using an inference

engine.

- Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.
- The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

It is a down-up approach, as it moves from bottom to top.

- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.
- Forward-chaining approach is also called as datadriven as we reach to the goal using available data.
- Forward -chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Forward chaining example

- KB:
 - $\text{allergies}(X) \rightarrow \text{sneeze}(X)$
 - $\text{cat}(Y) \wedge \text{allergic-to-cats}(X) \rightarrow \text{allergies}(X)$
 - $\text{cat}(\text{Felix})$
 - $\text{allergic-to-cats}(\text{Lise})$
- Goal:
 - $\text{sneeze}(\text{Lise})$

Backward chaining

- **Backward-chaining** deduction using GMP is also **complete** for KBs containing **only Horn clauses**

- Proofs start with the goal query, find rules with that conclusion, and then prove each of the antecedents in the implication
- Keep going until you reach premises
- Avoid loops: check if new subgoal is already on the goal stack
- Avoid repeated work: check if new subgoal
 - Has already been proved true
 - Has already failed

Backward chaining example

- KB:
 - $\text{allergies}(X) \rightarrow \text{sneeze}(X)$
 - $\text{cat}(Y) \wedge \text{allergic-to-cats}(X) \rightarrow \text{allergies}(X)$
 - $\text{cat}(\text{Felix})$
 - $\text{allergic-to-cats}(\text{Lise})$
- Goal:
 - $\text{sneeze}(\text{Lise})$

Forward vs. backward chaining

- FC is data-driven
 - Automatic, unconscious processing
 - E.g., object recognition, routine decisions
 - May do lots of work that is irrelevant to the goal
- BC is goal-driven, appropriate for problem-solving
 - Where are my keys? How do I get to my next class?

- Complexity of BC can be much less than linear in the size of the KB

Unit-IV

KNOWLEDGE REPRESENTATION

KNOWLEDGE REPRESENTATION:-

For the purpose of solving complex problems encountered in AI, we need both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge (facts) have been exploited in AI programs. In all variety of knowledge representations, we deal with two kinds of entities.

A. Facts: Truths in some relevant world. These are the things we want to represent.

B. Representations of facts in some chosen formalism. These are things we will actually be able to manipulate.

One way to think of structuring these entities is at two levels: (a) the knowledge level, at which facts are described, and (b) the symbol level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

The facts and representations are linked with two-way mappings. This link is called representation mappings. The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations to facts.

One common representation is natural language (particularly English) sentences. Regardless of the representation for facts we use in a program, we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. We need mapping functions from English sentences to the representation we actually use and from it back to sentences.

Representations and Mappings

- In order to solve complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions.
- Knowledge and Representation are two distinct entities. They play central but distinguishable roles in the intelligent system.
- Knowledge is a description of the world. It determines a system's competence by

what it knows.

- Moreover, Representation is the way knowledge is encoded. It defines a system's performance in doing something.
- Different types of knowledge require different kinds of representation.

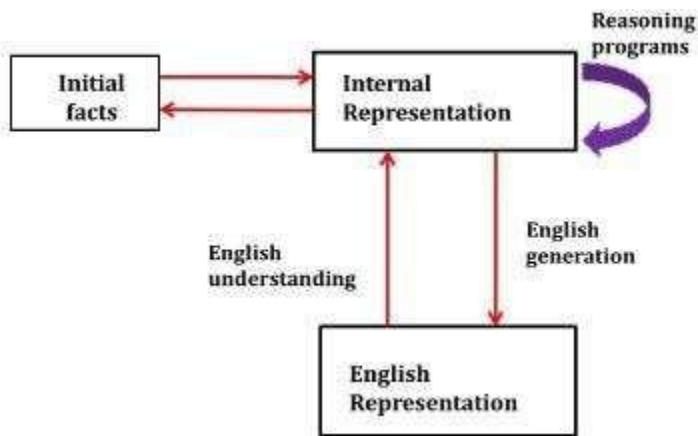


Fig: Mapping between Facts and Representations

The Knowledge Representation models/mechanisms are often based on:

- Logic
- Rules
- Frames
- Semantic Net

Knowledge is categorized into two major types:

1. Tacit corresponds to “informal” or “implicit”
 - Exists within a human being;
 - It is embodied.
 - Difficult to articulate formally.
 - Difficult to communicate or share.
 - Moreover, Hard to steal or copy.
 - Drawn from experience, action, subjective insight
2. Explicit formal type of knowledge, Explicit
 - Explicit knowledge
 - Exists outside a human being;
 - It is embedded.
 - Can be articulated formally.
 - Also, Can be shared, copied, processed and stored.
 - So, Easy to steal or copy
 - Drawn from the artifact of some type as a principle, procedure, process, concepts. A variety of ways of representing knowledge have been exploited in AI programs.

There are two different kinds of entities, we are dealing with.

1. Facts: Truth in some relevant world. Things we want to represent.
2. Also, Representation of facts in some chosen formalism. Things we will actually be able to manipulate.

These entities structured at two levels:

1. The knowledge level, at which facts described.
2. Moreover, The symbol level, at which representation of objects defined in terms of symbols that can manipulate by programs

Framework of Knowledge Representation

- The computer requires a well-defined problem description to process and provide a well- defined acceptable solution.

- Moreover, To collect fragments of knowledge we need first to formulate a description in our spoken language and then represent it in formal language so that computer can understand.
- Also, The computer can then use an algorithm to compute an answer. So, This process illustrated as,

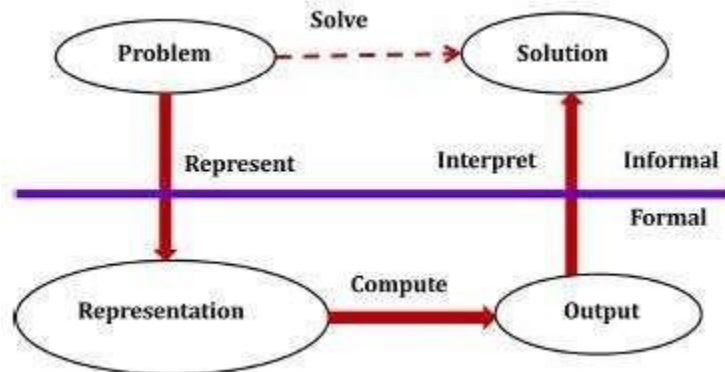


Fig: Knowledge Representation Framework

The steps are:

- The informal formalism of the problem takes place first.
- It then represented formally and the computer produces an output.
- This output can then represented in an informally described solution that user understands or checks for consistency.

The Problem solving requires,

- Formal knowledge representation, and
- Moreover, Conversion of informal knowledge to a formal knowledge that is the conversion of implicit knowledge to explicit knowledge.

Mapping between Facts and Representation

- Knowledge is a collection of facts from some domain.
- Also, We need a representation of “facts” that can manipulate by a program.
- Moreover, Normal English is insufficient, too hard currently for a computer program to draw inferences in natural languages.
- Thus some symbolic representation is necessary.

A good knowledge representation enables fast and accurate access to knowledge and understanding of the content.

A knowledge representation system should have following properties.

1. Representational Adequacy
 - The ability to represent all kinds of knowledge that are needed in that domain.
2. Inferential Adequacy
 - Also, The ability to manipulate the representational structures to derive new structures corresponding to new knowledge inferred from old.
3. Inferential Efficiency
 - The ability to incorporate additional information into the knowledge structure that can be used to focus the attention of the inference mechanisms in the most promising direction.

4. Acquisitional Efficiency

- Moreover, The ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

Knowledge Representation Schemes Relational Knowledge

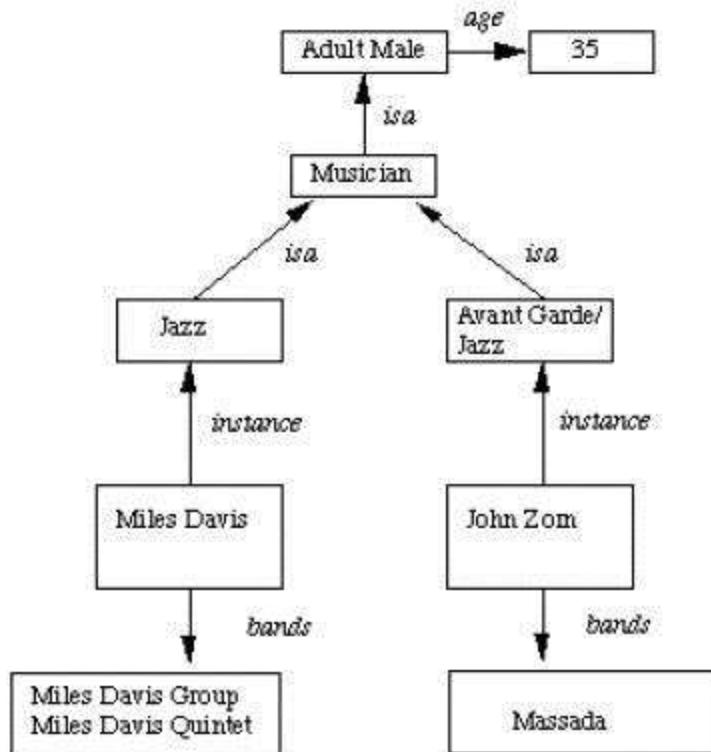
- The simplest way to represent declarative facts is a set of relations of the same sort used in the database system.
- Provides a framework to compare two objects based on equivalent attributes. o Any instance in which two different objects are compared is a relational type of knowledge.
- The table below shows a simple way to store facts.
 - Also, The facts about a set of objects are put systematically in columns.
 - This representation provides little opportunity for inference.

Player	Height	Weight	Bats - Throws
Aaron	6-0	180	Right - Right
Mays	5-10	170	Right - Right
Ruth	6-2	215	Left - Left
Williams	6-3	205	Left - Right

- Given the facts, it is not possible to answer a simple question such as: “Who is the heaviest player?”
- Also, But if a procedure for finding the heaviest player is provided, then these facts will enable that procedure to compute an answer.
- Moreover, We can ask things like who “bats – left” and “throws – right”.

Inheritable Knowledge

- Here the knowledge elements inherit attributes from their parents.
- The knowledge embodied in the design hierarchies found in the functional, physical and process domains.
- Within the hierarchy, elements inherit attributes from their parents, but in many cases, not all attributes of the parent elements prescribed to the child elements.
- Also, The inheritance is a powerful form of inference, but not adequate.
- Moreover, The basic KR (Knowledge Representation) needs to augment with inference mechanism.
- Property inheritance: The objects or elements of specific classes inherit attributes and values from more general classes.
- So, The classes organized in a generalized hierarchy.



- Boxed nodes — objects and values of attributes of objects.
- Arrows — the point from object to its value.
- This structure is known as a slot and filler structure, semantic network or a collection of frames.

The steps to retrieve a value for an attribute of an instance object:

1. Find the object in the knowledge base
2. If there is a value for the attribute report it
3. Otherwise look for a value of an instance, if none fail
4. Also, Go to that node and find a value for the attribute and then report it
5. Otherwise, search through using is until a value is found for the attribute.

Inferential Knowledge

- This knowledge generates new information from the given information.
- This new information does not require further data gathering form source but does require analysis of the given information to generate new knowledge.
- Example: given a set of relations and values, one may infer other values or relations. A predicate logic (a mathematical deduction) used to infer from a set of attributes. Moreover, Inference through predicate logic uses a set of logical operations to relate individual data.
- Represent knowledge as formal logic: All dogs have tails $\forall x: dog(x) \rightarrow hastail(x)$
- Advantages:
 - A set of strict rules.
 - Can use to derive more facts.
 - Also, Truths of new statements can be verified.

- Guaranteed correctness.
- So, Many inference procedures available to implement standard rules of logic popular in AI systems. e.g Automated theorem proving.

Procedural Knowledge

- A representation in which the control information, to use the knowledge, embedded in the knowledge itself. For example, computer programs, directions, and recipes; these indicate specific use or implementation;
- Moreover, Knowledge encoded in some procedures, small programs that know how to do specific things, how to proceed.
- Advantages:
 - Heuristic or domain-specific knowledge can represent.
 - Moreover, Extended logical inferences, such as default reasoning facilitated.
 - Also, Side effects of actions may model. Some rules may become false in time. Keeping track of this in large systems may be tricky.
- Disadvantages:
 - Completeness — not all cases may represent.
 - Consistency — not all deductions may be correct. e.g If we know that Fred is a bird we might deduce that Fred can fly. Later we might discover that Fred is an emu.
 - Modularity sacrificed. Changes in knowledge base might have far-reaching effects.
 - Cumbersome control information.

1. Everyone is loyal to someone.
 - $\forall x: \exists y: \text{loyalto}(x, y)$
2. People only try to assassinate rulers they are not loyal to.
 - $\forall x: \forall y: \text{person}(x) \wedge \text{ruler}(y) \wedge \text{tryassassinate}(x, y)$
 - $\rightarrow \neg \text{loyalto}(x, y)$
3. Marcus tried to assassinate Caesar.
 - $\text{tryassassinate}(\text{Marcus}, \text{Caesar})$

Now suppose if we want to use these statements to answer the question: ***Was Marcus loyal to Caesar?***

Also, Now let's try to produce a formal proof, reasoning backward from the desired goal: $\neg \text{loyalto}(\text{Marcus}, \text{Caesar})$

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can, in turn, transformed, and so on, until there are no unsatisfied goals remaining.

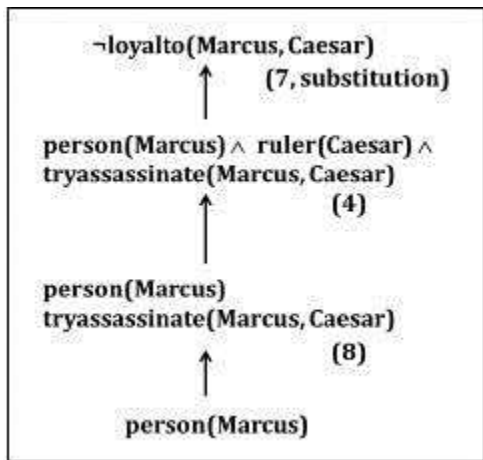


Figure: An attempt to prove $\neg\text{loyalto}(\text{Marcus}, \text{Caesar})$.

- The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. Also, We need to add the representation of another fact to our system, namely: $\forall \text{man}(x) \rightarrow \text{person}(x)$
- Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.
- Moreover, From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:
 1. Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
 2. Also, There is often a choice of how to represent the knowledge. Simple representations are desirable, but they may exclude certain kinds of reasoning.
 3. Similarly, Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively. Moreover, It is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention.

Weak Slot and Filler Structures

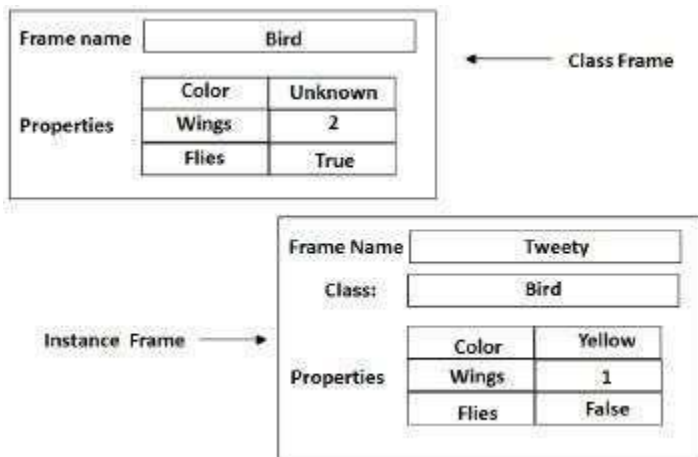
Evolution Frames

- As seen in the previous example, there are certain problems which are difficult to solve with Semantic Nets.
- Although there is no clear distinction between a semantic net and frame system, more structured the system is, more likely it is to be termed as a frame system.

- A frame is a collection of attributes (called slots) and associated values that describe some entities in the world. Sometimes a frame describes an entity in some absolute sense;
- Sometimes it represents the entity from a particular point of view only.
- A single frame taken alone is rarely useful; we build frame systems out of collections of frames that connected to each other by virtue of the fact that the value of an attribute of one frame may be another frame.

Frames as Sets and Instances

- The set theory is a good basis for understanding frame systems.
- Each frame represents either a class (a set) or an instance (an element of class)
- Both ***isa*** and ***instance*** relations have inverse attributes, which we call subclasses & all instances.
- As a class represents a set, there are 2 kinds of attributes that can be associated with it.
 1. Its own attributes &
 2. Attributes that are to be inherited by each element of the set.



Frames as Sets and Instances

- Sometimes, the difference between a set and an individual instance may not be clear.
- Example: Team India is an instance of the class of Cricket Teams and can also think of as the set of players.
- Now the problem is if we present Team India as a subclass of Cricket teams, then Indian players automatically become part of all the teams, which is not true.
- So, we can make Team India a subclass of class called Cricket Players.
- To do this we need to differentiate between regular classes and meta-classes.
- Regular Classes are those whose elements are individual entities whereas Meta-classes are those special classes whose elements are themselves, classes.
- The most basic meta-class is the class *CLASS*.
- It represents the set of all classes.
- All classes are instances of it, either directly or through one of its subclasses.
- The class *CLASS* introduces the attribute cardinality, which is to inherited by all instances of *CLASS*. Cardinality stands for the number.

Other ways of Relating Classes to Each Other

- We have discussed that a class1 can be a subset of class2.
- If Class2 is a meta-class then Class1 can be an instance of Class2.
- Another way is the **mutually-disjoint-with** relationship, which relates a class to one or more other classes that guaranteed to have no elements in common with it.
- Another one is, **is-covered-by** which relates a class to a set of subclasses, the union of which is equal to it.
- If a class is-covered-by a set S of mutually disjoint classes, then S called a partition of the class.

Slots as Full-Fledged Objects (Frames)

Till now we have used attributes as slots, but now we will represent attributes explicitly and describe their properties.

Some of the properties we would like to be able to represent and use in reasoning include,

- The class to which the attribute can attach.
- Constraints on either the type or the value of the attribute.
- A default value for the attribute. Rules for inheriting values for the attribute.

- To be able to represent these attributes of attributes, we need to describe attributes (slots) as frames.

- These frames will organize into an *isa* hierarchy, just as any other frames, and that hierarchy can then be used to support inheritance of values for attributes of slots.
- Now let us formalize what is a slot. A slot here is a relation.
- It maps from elements of its domain (the classes for which it makes sense) to elements of its range (its possible values).
- A relation is a set of ordered pairs.
- Thus it makes sense to say that relation R1 is a subset of another relation R2.
- In that case, R1 is a specialization of R2. Since a slot is a set, the set of all slots, which we will call SLOT, is a meta-class.
- Its instances are slots, which may have sub-slots.

Frame Example

In this example, the frames Person, Adult-Male, ML-Baseball-Player (corresponding to major league baseball players), Pitcher, and ML-Baseball-Team (for major league baseball team) are all classes.

```

Person
  isa : Mammal
  cardinality : 6,000,000,000
  * handed : Right
Adult-Male
  isa : Person
  cardinality : 2,000,000,000
  * height : 5-10
ML-Baseball-Player
  isa : Adult-Male
  cardinality : 624
  * height : 6-1
  * bats : equal to handed
  * batting-average : .252
  * team :
  * uniform-color :
Fielder
  isa : ML-Baseball-Player
  cardinality : 376
  * batting-average : .262
Pee-Wee-Reese
  instance : Fielder
  height : 5-10
  bats : Right
  batting-average : .309
  team : Brooklyn-Dodgers
  uniform-color : Blue
ML-Baseball-Team
  isa : Team
  cardinality : 26
  * team-size : 24
  * manager :
Brooklyn-Dodgers
  instance : ML-Baseball-Team
  team-size : 24
  manager : Leo-Durocher
  players : {Pee-Wee-Reese,...}

```

- The frames Pee-Wee-Reese and Brooklyn-Dodgers are instances.
- The *isa* relation that we have been using without a precise definition is, in fact, the subset relation. The set of adult males is a subset of the set of people.
- The set of major league baseball players is a subset of the set of adult males, and so forth.
- Our instance relation corresponds to the relation element-of. Pee Wee Reese is an element of the set of fielders.
- Thus he is also an element of all of the supersets of fielders, including major league baseball players and people. The transitivity of *isa* follows directly from the transitivity of the subset relation.

- Both the isa and instance relations have inverse attributes, which we call subclasses and all instances.
- Because a class represents a set, there are two kinds of attributes that can associate with it.
- Some attributes are about the set itself, and some attributes are to inherited by each element of the set.
- We indicate the difference between these two by prefixing the latter with an asterisk (*).
- For example, consider the class ML-Baseball-Player, we have shown only two properties of it as a set: It a subset of the set of adult males. And it has cardinality 624.
- We have listed five properties that all major league baseball players have (height, bats, batting average, team, and uniform-color), and we have specified default values for the first three of them.
- By providing both kinds of slots, we allow both classes to define a set of objects and to describe a prototypical object of the set.
- Frames are useful for representing objects that are typical of stereotypical situations.
- The situation like the structure of complex physical objects, visual scenes, etc.
- A commonsense knowledge can represent using default values if no other value exists. Commonsense is generally used in the absence of specific knowledge.

Semantic Nets

- Inheritance property can represent using **isa** and **instance**
- Monotonic Inheritance can perform substantially more efficiently with such structures than with pure logic, and non-monotonic inheritance is also easily supported.
- The reason that makes Inheritance easy is that the knowledge in slot and filler systems is structured as a set of entities and their attributes.

These structures turn out to be useful as,

- It indexes assertions by the entities they describe. As a result, retrieving the value for an attribute of an entity is fast.
- Moreover, It makes easy to describe properties of relations. To do this in a purely logical system requires higher-order mechanisms.
- It is a form of object-oriented programming and has the advantages that such systems normally include modularity and ease of viewing by people.

Here we would describe two views of this kind of structure – Semantic Nets & Frames.

Semantic Nets

- There are different approaches to knowledge representation include semantic net, frames, and script.
- The semantic net describes both objects and events.
- In a semantic net, information represented as a set of nodes connected to each other by a set of labeled arcs, which represents relationships among the nodes.
- It is a directed graph consisting of vertices which represent concepts and edges which represent semantic relations between the concepts.
- It is also known as associative net due to the association of one node with other.
- The main idea is that the meaning of the concept comes from the ways in

which it connected to other concepts.

- We can use inheritance to derive additional relations.

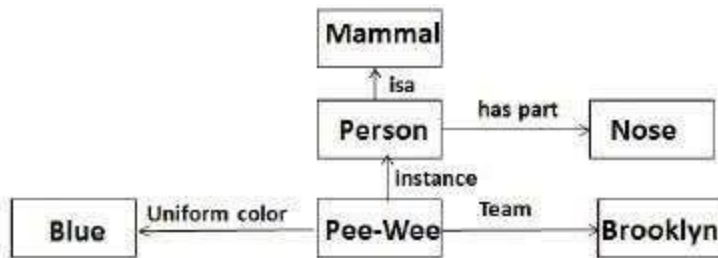


Figure: A Semantic Network

Intersection Search Semantic Nets

- We try to find relationships among objects by spreading activation out from each of two nodes. And seeing where the activation meets.
- Using this we can answer the questions like, what is the relation between India and Blue.
- It takes advantage of the entity-based organization of knowledge that slot and filler representation provides.

Representing Non-binary Predicates Semantic Nets

- Simple binary predicates like `isa(Person, Mammal)` can represent easily by semantic nets but other non-binary predicates can also represent by using general-purpose predicates such as *isa* and *instance*.
- Three or even more place predicates can also convert to a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe a relationship to this new object.

Conceptual Dependency

Introduction to Strong Slot and Filler Structures

- The main problem with semantic networks and frames is that they lack formality; there is no specific guideline on how to use the representations.
- In frame when things change, we need to modify all frames that are relevant – this can be time-consuming.
- Strong slot and filler structures typically represent links between objects according to more rigid rules, specific notions of what types of object and relations between them are provided and represent knowledge about common situations.
- Moreover, We have types of strong slot and filler structures:
 1. Conceptual Dependency (CD)
 2. Scripts
 3. Cyc

Conceptual Dependency (CD)

Conceptual Dependency originally developed to represent knowledge acquired from natural language input.

The goals of this theory are:

- To help in the drawing of the inference from sentences.
- To be independent of the words used in the original input.
- That is to say: For any 2 (or more) sentences that are identical in meaning there

should be only one representation of that meaning.

Moreover, It has used by many programs that portend to understand English (MARGIE, SAM, PAM).

Conceptual Dependency (CD) provides:

- A structure into which nodes representing information can be placed.
- Also, A specific set of primitives.
- A given level of granularity.

Sentences are represented as a series of diagrams depicting actions using both abstract and real physical situations.

- The agent and the objects represented.
- Moreover, The actions are built up from a set of primitive acts which can modify by tense.

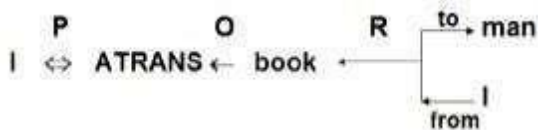
CD is based on events and actions. Every event (if applicable) has:

- an ACTOR or an ACTION performed by the Actor
- Also, an OBJECT that the action performs on
- A DIRECTION in which that action is oriented

These are represented as slots and fillers. In English sentences, many of these attributes left out.

A Simple Conceptual Dependency Representation

For the sentences, "I have a book to the man" CD representation is as follows:



Where the symbols have the following meaning.

- Arrows indicate directions of dependency.
- Moreover, The double arrow indicates the two-way link between actor and action.
- O — for the object case relation
- R – for the recipient case relation
- P – for past tense
- D – destination

Primitive Acts of Conceptual Dependency Theory

ATRANS

- Transfer of an abstract relationship (i.e. give) PTRANS
- Transfer of the physical location of an object (e.g., go) PROPEL
- Also, Application of physical force to an object (e.g. push) MOVE
- Moreover, Movement of a body part by its owner (e.g. kick) GRASP
- Grasping of an object by an action (e.g. throw) INGEST
- Ingesting of an object by an animal (e.g. eat) EXPEL
- Expulsion of something from the body of an animal (e.g. cry) MTRANS

- Transfer of mental information (e.g. tell) MBUILD
- Building new information out of old (e.g. decide) SPEAK

- Producing of sounds (e.g. say) ATTEND
- Focusing of a sense organ toward a stimulus (e.g. listen)

There are four conceptual categories. These are,

ACT

- Actions {one of the CD primitives}

PP

- Also, Objects {picture producers}

AA

- Modifiers of actions {action aiders}

PA

- Modifiers of PP's {picture aiders}

Advantages of Conceptual Dependency

- Using these primitives involves fewer inference rules.
- So, Many inference rules already represented in CD structure.
- Moreover, The holes in the initial structure help to focus on the points still to established.

Disadvantages of Conceptual Dependency

- Knowledge must decompose into fairly low-level primitives.
- Impossible or difficult to find the correct set of primitives.
- Also, A lot of inference may still require.
- Representations can be complex even for relatively simple actions.
- Consider: Dave bet Frank five pounds that Wales would win the Rugby World Cup.
- Moreover, Complex representations require a lot of storage.

Scripts

Scripts Strong Slot

- A script is a structure that prescribes a set of circumstances which could be expected to follow on from one another.
- It is similar to a thought sequence or a chain of situations which could be anticipated.
- It could be considered to consist of a number of slots or frames but with more specialized roles.

Scripts are beneficial because:

- Events tend to occur in known runs or patterns.
- Causal relationships between events exist.
- Entry conditions exist which allow an event to take place
- Prerequisites exist for events taking place. E.g. when a student progresses through a degree scheme or when a purchaser buys a house.

Script Components

Each script contains the following main components.

- Entry Conditions: Must be satisfied before events in the script can occur.
- Results: Conditions that will be true after events in script occur.
- Props: Slots representing objects involved in the events.

- Roles: Persons involved in the events.
- Track: the Specific variation on the more general pattern in the script. Different tracks may share many components of the same script but not all.

- Scenes: The sequence of events that occur. Events represented in conceptual dependency form.

Advantages and Disadvantages of Script

Advantages

- Capable of predicting implicit events
- Single coherent interpretation may be build up from a collection of observations. Disadvantage
- More specific (inflexible) and less general than frames.
- Not suitable to represent all kinds of knowledge.

To deal with inflexibility, smaller modules called memory organization packets (MOP) can combine in a way that appropriates for the situation.

Script Example

Script : Play in theater	Various Scenes
Track: Play in Theater Props: • Tickets • Seat • Play Roles: • Person (who wants to see a play) – P • Ticket distributor – TD • Ticket checker – TC Entry Conditions: • P wants to see a play • P has a money Results: • P saw a play • P has less money • P is happy (optional if he liked the play)	Scene 1: Going to theater • P PTRANS P into theater • P ATTEND eyes to ticket counter
	Scene 2: Buying ticket • P PTRANS P to ticket counter • P MTRANS (need a ticket) to TD • TD ATRANS ticket to P
	Scene 3: Going inside hall of theater and sitting on a seat • P PTRANS P into Hall of theater • TC ATTEND eyes on ticket POSS_by P • TC MTRANS (showed seat) to P • P PTRANS P to seat • P MOVES P to sitting position
	Scene 4: Watching a play • P ATTEND eyes on play • P MBUILD (good moments) from play
	Scene 5: Exiting • P PTRANS P out of Hall and theater

- It must activate based on its significance.
- If the topic important, then the script should open.
- If a topic just mentioned, then a pointer to that script could hold.
- For example, given “John enjoyed the play in theater”, a script “Play in Theater” suggested above invoke.
- All implicit

questions can answer correctly. Here the significance of this script is high.

- Did John go to the theater?

- Also, Did he buy the ticket?
- Did he have money?

If we have a sentence like "John went to the theater to pick his daughter", then invoking this script will lead to many wrong answers.

- Here significance of the script theater is less.

Unit V

Prolog

Prolog is a **logic programming** language, first invented by researchers in France around 1972. The name Prolog comes from the phrase "**programming in logic**". In the logic programming paradigm, a program consists of a set of logical statements. In fact Prolog is actually a simple kind of theorem prover: given a logical formula, it attempts to determine whether it can be satisfied. Prolog's syntax and semantics are closely related to [first-order logic](#), which you may have studied in another class.

To be more precise, Prolog is based on a Turing-complete subset of first-order logic in which every program is a sequence of [Horn clauses](#). However, we will not deeply study the logical foundation of Prolog in this course. Our focus will be on using the language, i.e. writing programs to accomplish our goals.

As I mentioned in the lecture, in our class we will use [SWI-Prolog](#), which you should install on your own machine now. To start the Prolog REPL, run the command "swipl":

```
$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is
free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-
prolog.org
For built-in help, use ?- help(Topic). or ?-
apropos(Word) .

1 ?-
```

You'll want to edit your Prolog code in some development environment. I recommend [Visual Studio Code](#). If you use it, I recommend installing the extension called "Prolog" (the author is Peng Lv). I also recommend

configuring Code so that files with the extension ".pl" are recognized as Prolog (by default, Code thinks these are Perl files). To do that, open Settings, then find the Files : Associations setting, where you can add a mapping from "*.pl" to "prolog".

terms

Data in Prolog is represented as **terms**. For now, a term may be

- an **atom**, which is a string beginning with a **lowercase letter** and containing letters, digits and '_'. Examples: red, blue, dog, cat, house. All atoms are distinct, i.e. not equal to each other.
- a **variable**, which begins with an **uppercase letter** and may contain letters, digits, and '_'. Examples: X, Y, Color.

We will learn about other kinds of terms soon.

equality

The built-in predicate '=' is true when two terms are equal.

```
?- red = red.  
true.
```

```
?- red = blue.  
false.
```

`dif(X, Y)` is true when X and Y are **different**, i.e. not equal.

```
?- dif(red, blue).  
true.
```

```
?- dif(red, red).  
false.
```

Boolean operators

Prolog includes these Boolean operators:

- `,` (Boolean AND)
- `;` (Boolean OR)

```
?- red = blue, blue = blue.  
false.
```

```
?- red = blue; blue = blue.
```

```
true.
```

queries with variables

A Prolog query may include variables. Prolog attempts to prove that the query is true for **some values** of the given variables. In other words, the variables are **existentially quantified**. If the query can be true, Prolog reports the values that make it so.

```
?- X = red.  
X = red.
```

```
?- X = red, X = blue.  
false.
```

```
?- (X = red; X = blue), (Y = green; Y = yellow).  
X = red,  
Y = green ;  
X = red,  
Y = yellow ;  
X = blue,  
Y = green ;  
X = blue,  
Y = yellow.
```

A query may succeed multiple times, as in the last query above. When there may be multiple solutions, Prolog prints the first solution, then waits for user input. At this point you have two choices:

- Type ';' to ask Prolog to look for another solution. It will continue searching, and will print the next solution if it can find one, otherwise 'false'.
- Type '.' or Enter if you don't want to look for more solutions, and want to return to the REPL prompt.

Notice that Prolog solved this query by performing a **depth-first search**. First it saw the subgoal '(X = red; X = blue)'. This is a **disjunction**, i.e. two subgoals separated by the OR (;) operator. Any disjunction in Prolog introduces a **choice point**, i.e. a place where Prolog's search can go in one of two directions. So first Prolog assumes that X = red. Then it considers each of the values (Y = green; Y = yellow), and reports a solution for each of them:

```
X = red,  
Y = green ;  
X = red,
```

```
Y = yellow ;
```

After that, Prolog **backtracks** to the choice point for X. It now chooses the value X = blue, and the proceeds again to the subgoal (Y = green; Y = yellow), reporting two more solutions:

```
X = blue,  
Y = green ;  
X = blue,  
Y = yellow.
```

Here are some more queries involving multiple variables:

```
?- (X = blue; X = red), (Y = red; Y = green), X = Y.  
X = Y, Y = red
```

```
?- (X = blue; X = red), (Y = red; Y = green), dif(X, Y).  
X = blue,  
Y = red ;  
X = blue,  
Y = green ;  
X = red,  
Y = green.
```