MODULE-03

Knowledge Representation - Knowledge-Based Agents, The Wumpus World , Logic, Propositional Logic, Propositional Theorem Proving, Effective Propositional Model Checking, Agents Based on Propositional Logic, First-Order Logic-Syntax and Semantics of First-Order Logic, Using First-Order Logic, Unification and Lifting ,Forward Chaining, Backward Chaining.

3.Knowledge Representation:

Humans are best at understanding, reasoning, and interpreting knowledge. Human knows things, which is knowledge and as per their knowledge they perform various actions in the real world. **But how machines do all these things comes under knowledge representation and reasoning**. Hence we can describe Knowledge representation as following:

- Knowledge representation and reasoning (KR, KRR) is the part of Artificial intelligence which concerned with AI agents thinking and how thinking contributes to intelligent behaviour of agents.
- It is responsible for representing information about the real world so that a computer can understand and can utilize this knowledge to solve the complex real world problems such as diagnosis a medical condition or communicating with humans in natural language.
- It is also a way which describes how we can represent knowledge in artificial intelligence. Knowledge representation is not just storing data into some database, but it also enables an intelligent machine to learn from that knowledge and experiences so that it can behave intelligently like a human.

What to Represent:

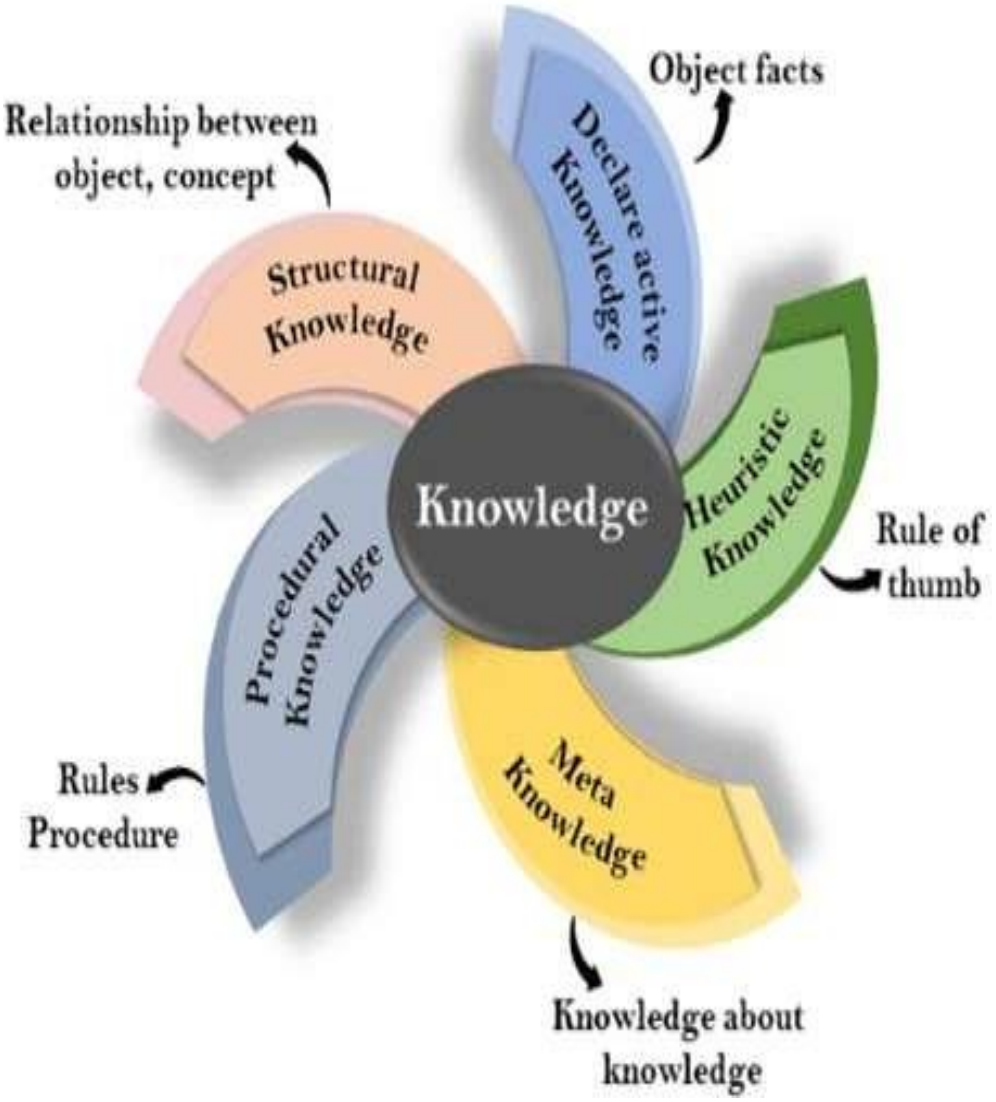
Following are the kind of knowledge which needs to be represented in AI systems:

- **Object:** All the facts about objects in our world domain. E.g., Guitars contains strings, trumpets are brass instruments.
- **Events:** Events are the actions which occur in our world.
- **Performance:** It describe behaviour which involves knowledge
- **Meta-knowledge:** It is knowledge about what we know.
- **Facts:** Facts are the truths about the real world and what we represent.
- **Knowledge-Base:** The central component of the knowledge- based agents is the knowledge base. It is represented as KB. The Knowledgebase is a group of the Sentences (Here, sentences are used as a technical term and not identical with the English language).

Knowledge: Knowledge is awareness or familiarity gained by experiences of facts, data, and situations. Following are the types of knowledge in artificial intelligence:

Types of knowledge

Following are the various types of knowledge:



📚 Types of Knowledge in AI

1️⃣ Declarative Knowledge (Knowing *What*)

- **Meaning**: Knowledge **about facts, objects, and concepts**.
- **Also called**: Descriptive Knowledge.
- **Expressed in**: Sentences like "The sky is blue" or "Ananya is a BCA student".
- **Example**:
  - "All BCA students study Computer Science."
  - "The capital of Karnataka is Bengaluru."
- ✅ Used to **describe** something.
- 🧠 It's *easier to express and understand* than how to *do* something.

2️⃣ Procedural Knowledge (Knowing *How*)

- **Meaning**: Knowledge about **how to do things** — steps, actions, and methods.
- **Also called**: Imperative Knowledge.
- **Includes**: Rules, strategies, procedures.
- **Example**:
  - "To calculate average marks: add all marks and divide by number of subjects."
  - "If a student fails in 3 or more subjects, mark them as 'Fail'."
- ✅ Used to **solve problems** or perform actions.
- 🛠️ It's **action-based** and **task-specific**.

3️⃣ Meta-Knowledge (Knowing *About Knowledge*)

- **Meaning**: Knowledge **about** other types of knowledge.
- 🧠 It helps in organizing, choosing, and managing what knowledge to apply in what situation.
- **Example**:
  - "Declarative knowledge is used to describe facts, while procedural knowledge is used for solving tasks."
  - "Knowing that a certain problem needs heuristic rather than rule-based methods."

4️⃣ Heuristic Knowledge (Rules of Thumb)

- **Meaning**: Expert-level knowledge based on **experience** and **intuition**.
- **Also called**: Common sense or "educated guess" rules.
- **Not always 100% accurate** but works well in many cases.
- **Example**:
  - "If most students score low, simplify the test."
  - "In AI games, if your opponent always moves left, block left."
- ⚠️ Not guaranteed to be correct, but often **very useful** for making decisions.

Type of Knowledge	What it Means	Example
Declarative	Knowing *what*	"Ananya is a BCA student."
Procedural	Knowing *how*	"Add marks, then divide by total subjects."
Meta-Knowledge	Knowing *about knowledge*	"Declarative is for facts, procedural for tasks."
Heuristic Knowledge	Based on experience (rules of thumb)	"If unsure, pick the most common answer."

Knowledge Representation (KR) in **Artificial Intelligence (AI)** refers to the way in which intelligent systems store, organize, and utilize knowledge about the world to make decisions, solve problems, and interact intelligently.

Knowledge Representation is the field of AI dedicated to representing information about the real world in a form that a computer system can understand, interpret, and use to simulate human-like intelligence.

Knowledge Representation (KR) in AI is the method of encoding information about the world into a format that a computer system can use to solve complex tasks such as reasoning, learning, and decision-making.

Example: Real-World Example: College Student Course Recommendation System.

Let's consider a college student using an AI-based Course Recommendation System.

Knowledge Represented:

1. **Student Profile**
    - Name: Priya
    - Branch: BCA
    - Year: 2nd Year
    - Skills: Python, HTML, MySQL
    - Interests: Web Development, Data Science
    - Past Courses Taken: HTML Basics, Python for Beginners
2. **Course Knowledge Base**
    - Web Design using Bootstrap (Skill: HTML/CSS)
    - Machine Learning with Python (Skill: Python, Math)
    - MySQL Advanced (Skill: MySQL)
    - Cybersecurity Essentials (Skill: Basic Networking)
3. **Rules for Recommendation (Using Rule-Based KR)**
    - IF student knows Python AND interested in Data Science → RECOMMEND Machine Learning with Python
    - IF student knows HTML → RECOMMEND Web Design using Bootstrap
    - IF student has taken MySQL → RECOMMEND MySQL Advanced

3.1 Knowledge-Based Agents

A Knowledge-Based Agent in Artificial Intelligence is an intelligent agent that uses a knowledge base (KB) to make decisions and perform reasoning.

The knowledge base is a central repository of facts and rules about the world, represented in a structured form (such as first-order logic or propositional logic).

1. **KnowledgeBase(KB):**
   Contains facts and rules about the environment.
2. **InferenceEngine:**
   Applies logical reasoning to derive new facts or make decisions.
3. **Perception → Reasoning → Action Cycle:**

- **Perceives** the environment.
- **Updates** the knowledge base.
- **Infers** new knowledge using logic.
- **Acts** based on derived conclusions.

Structure of a Knowledge-Based Agent:

1. **Knowledge Base (KB):** Stores information (facts + rules).
2. **Inference Engine:** Draws logical conclusions from the KB.
3. **Percept Processing Unit:** Converts sensor data to logical facts.
4. **Action Selection Unit:** Chooses actions based on inference.

Example:

Suppose a KB has:

- Rule: *If it is raining, then the ground is wet.*
- Fact: *It is raining.*

The agent can infer:

- *The ground is wet* and act accordingly (e.g., carry an umbrella).

Applications of Knowledge-Based Agents:

1. **Medical Diagnosis Systems**
   - Used to identify diseases based on symptoms using expert-level knowledge.
   - Example: MYCIN (an early expert system for diagnosing bacterial infections).
2. **Legal Reasoning Systems**
   - Help in analyzing legal cases by applying rules and precedents.
3. **Customer Support (Chatbots)**
   - Intelligent chatbots that provide answers based on a knowledge base.
4. **Expert Systems**
   - Used in domains like agriculture, engineering, finance for decision-making.

5. **Robotics**
    - Robots use KB to navigate and interact with the environment intelligently.
6. **Game Playing**
    - Strategy-based games where decisions are made based on rules and facts.
7. **Search Engines & Recommendation Systems**
    - Enhance results by reasoning over user queries and existing data.

Advantages of Knowledge-Based Agents:

1. **Human-Like Reasoning**
    - Can simulate expert-level thinking using logic and inference.
2. **Modularity**
    - Easy to update or extend the knowledge base without rewriting the whole system.
3. **Transparency**
    - Reasoning can be explained (traceable logic behind decisions).
4. **Reusable Knowledge**
    - Knowledge can be reused across multiple domains or applications.
5. **Decision Support**
    - Assists humans in complex decision-making tasks.

Disadvantages of Knowledge-Based Agents:

1. **Complex Knowledge Representation**
    - Representing real-world knowledge in formal logic is difficult and time-consuming.
2. **Scalability Issues**
    - Large knowledge bases can slow down reasoning processes.
3. **Manual Knowledge Acquisition**
    - Experts are needed to feed domain knowledge into the system, which is time-consuming and expensive.
4. **Limited Learning**
    - Traditional KB agents do not learn from experience (unlike machine learning models).
5. **Uncertainty Handling**
    - Struggles with vague or probabilistic data unless integrated with fuzzy logic or probabilistic models.

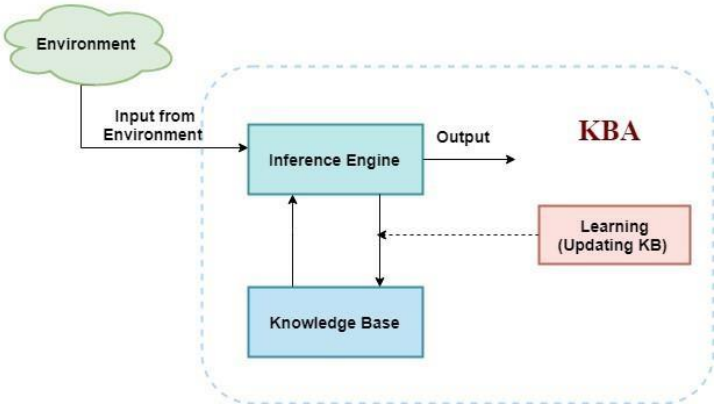
Knowledge-based agents are composed of two main parts:

- Knowledge-base and
- Inference system.

A knowledge-based agent must able to do the following:

- An agent should be able to represent states, actions, etc.
- An agent Should be able to incorporate new precepts
- An agent can update the internal representation of the world
- An agent can perform appropriate actions.

The architecture of knowledge-based agent:



- The above diagram is representing a generalized architecture for a knowledge-based agent. The knowledge-based agent (KBA) take input from the environment by perceiving the environment.
- The input is taken by the inference engine of the agent and which also communicate with KB to decide as per the knowledge store in KB. The learning element of KBA regularly updates the KB by learning new knowledge.
- Knowledge base: Knowledge-base is a central component of a knowledge-based agent, it is also known as KB. It is a collection of sentences. These sentences are expressed in a language which is called a knowledge representation language.

Inference system

Inference means deriving new sentences from old. Inference system allows us to add a new sentence to the knowledge base. A sentence is a proposition about the world. Inference system applies logical rules to the KB to deduce new information

College Student and Subjects

🧠 **Knowledge Base (KB) Stores Sentences Like:**

1. **TELL:** All BCA students study Computer Science.
2. **TELL:** Ananya is a BCA student.

Now, let's say you:

❓ **ASK: Does Ananya study Computer Science?**

Algorithm : A generic knowledge-based agent.

KB-AGENT(percept) returns an action
static:
  KB, a knowledge base
  t, a counter, initially 0

TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
action ← ASK(KB, MAKE-ACTION-QUERY(t))
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
t ← t + 1
return action

Explanation:

Time Step t = 0:

🔹 **Step 1: Agent receives a percept**

percept = "Ananya has enrolled in BCA"

📌 The agent does:

TELL(KB, MAKE-PERCEPT-SENTENCE("Ananya is a BCA student", 0))

✅ It stores this fact in the KB.

🔹 **Step 2: The agent asks:**

ASK(KB, MAKE-ACTION-QUERY(0))

💬 Something like:

"What subject does Ananya study?"

Now, the agent looks into the KB:

- KB knows: "All BCA students study Computer Science"
- KB knows: "Ananya is a BCA student"

🧠 Inference:

Since Ananya is a BCA student → She must study Computer Science

---

🔹 **Step 3: The agent stores the result:**

TELL(KB, MAKE-ACTION-SENTENCE("Ananya studies Computer Science", 0))

It remembers that it concluded this at time t = 0.

🔹 **Step 4: Move to the next time step:**

t ← t + 1

🔹 **Step 5: Return the final action:**

return "Ananya studies Computer Science"

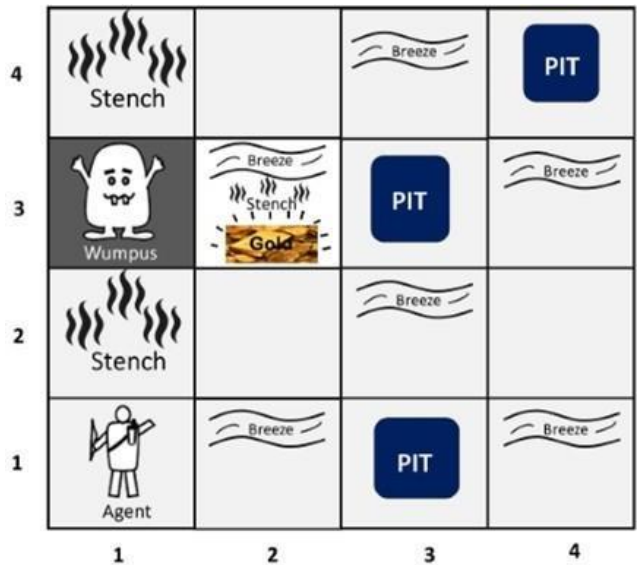
✅ The agent answers the question correctly using logical reasoning.

Step	Action
TELL	KB gets: "Ananya is a BCA student"
ASK	What does Ananya study?
INFERENCE	Uses fact: All BCA students study CS
TELL	Stores: "Ananya studies Computer Science"
RETURN	"Ananya studies Computer Science"

3.2 The Wumpus World in Artificial intelligence

- The Wumpus world is a simple world example to illustrate the worth of a knowledge based agent and to represent knowledge representation.

Problem definition: The Wumpus world is a cave which has 4/4 rooms connected with passageways. So there are total 16 rooms which are connected with each other. We have a knowledge-based agent who will go forward in this world. The cave has a room with a beast which is called Wumpus, who eats anyone who enters the room. The Wumpus can be shot by the agent, but the agent has a single arrow. In the Wumpus world, there are some Pits rooms which are bottomless, and if agent falls in Pits, then he will be stuck there forever. The exciting thing with this cave is that in one room there is a possibility of finding a heap of gold. So the agent goal is to find the gold and climb out the cave without fallen into Pits or eaten by Wumpus. The agent will get a reward if he comes out with gold, and he will get a penalty if eaten by Wumpus or falls in the pit.



There are also some components which can help the agent to navigate the cave. These components are given as follows:

- The rooms adjacent to the Wumpus room are smelly, so that it would have some stench.
- The room adjacent to PITs has a breeze, so if the agent reaches near to PIT, then he will perceive the breeze.
- There will be glitter in the room if and only if the room has gold.
- The Wumpus can be killed by the agent if the agent is facing to it.

PEAS description of Wumpus world:

Performance measure: +1000 reward points if the agent comes out of the

- cave with the gold.
-
- -1000 points penalty for being eaten by the Wumpus or falling into the pit.
- -1 for each action, and -10 for using an arrow.

The game ends if either agent dies or came out of the cave.

Environment:

- A 4*4 grid of rooms.
- The agent initially in room square [1, 1], facing toward the right.

Actuators:

- Left turn,
- Right turn
- Move forward
- Grab
- Release Shoot.

Sensors:

- The agent will perceive the stench if he is in the room adjacent to the Wumpus. (Not diagonally).
- The agent will perceive breeze if he is in the room directly adjacent to the Pit.
- The agent will perceive the glitter in the room where the gold is present.

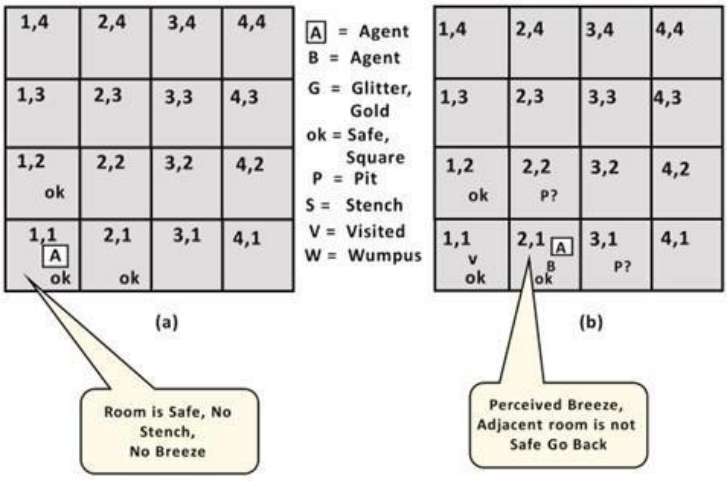
The Wumpus world Properties:

- Partially observable: The Wumpus world is partially observable because the agent can only perceive the close environment such as an adjacent room.
- Deterministic: It is deterministic, as the result and outcome of the world are already known.
- Sequential: The order is important, so it is sequential.
- Static: It is static as Wumpus and Pits are not moving.
- Discrete: The environment is discrete.
- Single agent: The environment is a single agent as we have one agent only and Wumpus is not considered as an agent.

Exploring the Wumpus world:

Agent's First step: Initially, the agent is in the first room or on the square [1,1], and we already know that this room is safe for the agent, so to represent on the below diagram (a) that room is safe we will add symbol OK. Symbol A is used to represent agent, symbol B for the breeze, G for Glitter or gold, V for the visited room, P for pits, W for Wumpus.

- At Room [1,1] agent does not feel any breeze or any Stench which means the adjacent squares are also OK

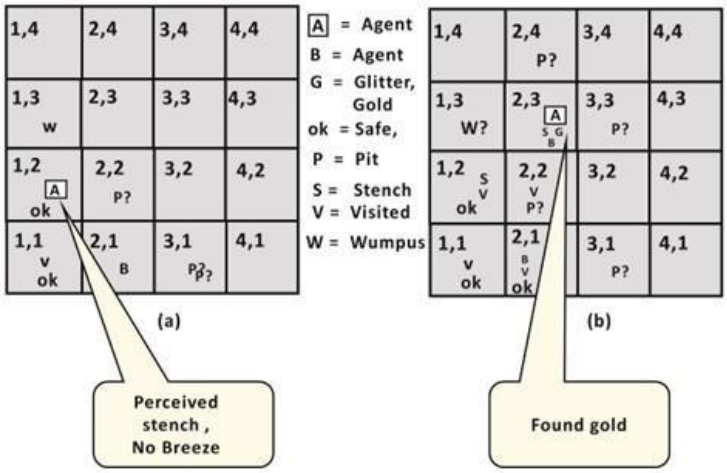


Agent's second Step:

- Now agent needs to move forward, so it will either move to [1, 2], or [2,1]. Let's suppose agent moves to the room [2, 1], at this room agent perceives some breeze which means Pit is around this room. The pit can be in [3, 1], or [2,2], so we will add symbol P? to say that, is this Pit room?
- The agent will go back to the [1, 1] room. The room [1,1], and [2,1] are visited by the agent, so we will use symbol V to represent the visited squares.

Agent's third step:

- At the third step, now agent will move to the room [1,2] which is OK. In the room [1,2] agent perceives a stench which means there must be a Wumpus nearby.
- But Wumpus cannot be in [2,2] (Agent had not detected any stench when he was at [2,1]). Therefore, agent infers that Wumpus is in the room [1,3], and in current state, there is no breeze which means in [2,2] there is no Pit and no Wumpus. So, it is safe, and we will mark it OK, and the agent moves further in [2,2].



Agent's fourth step:

- At room [2,2], here no stench and no breezes present so let's suppose agent decides to move to [2,3]. At room [2,3] agent perceives glitter, so it should grab the gold and climb out of the cave.

Wumpus world Algorithm:

Step 1:Initialize the agent's knowledge base (KB) with starting safe cell (0,0).

Step 2:Perceive the environment: check for stench, breeze, glitter (input from environment).

Step 3:Update KB based on perceptions: mark possible Wumpus (stench) or pits (breeze).

Step 4:Mark safe cells: if no breeze and no stench, surrounding cells are safe.

Step 5:Plan Move: choose an unvisited safe cell to move to.

Step 6:Move to the chosen cell and repeat perception.

Step 7:If glitter detected, grab gold and plan return path to start.

Step 8:If Wumpus location is certain, shoot the arrow to kill it.

Step 9:If stuck, backtrack to previous safe cells.

Step 10:Exit when gold is found or no safe moves left.

Example Program Wumpus world:

```
import random
# Define the environment
world = [
    ['S', '.', 'P', '.'],
    ['.', 'W', '.', 'P'],
    ['.', '.', '.', '.'],
    ['P', '.', '.', 'G']
]
# Legend:
# S = Start
# W = Wumpus
# P = Pit
# G = Gold
# . = Empty

# Agent starting position
agent_pos = [0, 0]
def perceive(x, y):
    perceptions = []
    if world[x][y] == 'G':
        perceptions.append('Glitter')
    if world[x][y] == 'W':
        perceptions.append('Stench')
    if world[x][y] == 'P':
        perceptions.append('Breeze')
    return perceptions
def move_agent():
    global agent_pos
    moves = []
    x, y = agent_pos
    if x > 0:
        moves.append((x-1, y))
    if x < 3:
        moves.append((x+1, y))
    if y > 0:
        moves.append((x, y-1))
    if y < 3:
        moves.append((x, y+1))
```

```
    agent_pos = random.choice(moves)
# Game Loop
for _ in range(10):  # Limit steps
    x, y = agent_pos
    perceptions = perceive(x, y)
    print(f"Agent is at ({x}, {y}) - Perceptions: {perceptions}")

    if 'Glitter' in perceptions:
        print("Agent found the Gold! 🏆")
        break
    if 'W' == world[x][y]:
        print("Agent was eaten by the Wumpus! 💀")
        break
    if 'P' == world[x][y]:
        print("Agent fell into a Pit! 💀")
        break
    move_agent()
```

OUTPUT:

Agent is at (0, 0) - Perceptions: []

Agent is at (1, 0) - Perceptions: []

Agent is at (2, 0) - Perceptions: []

Agent is at (2, 1) - Perceptions: []

Agent is at (2, 2) - Perceptions: []

Agent is at (3, 2) - Perceptions: []

Agent is at (3, 3) - Perceptions: ['Glitter']

Agent found the Gold! 🏆

Another output:

Agent is at (0, 0) - Perceptions: []

Agent is at (1, 0) - Perceptions: []

Agent is at (1, 1) - Perceptions: ['Stench']

Agent was eaten by the Wumpus! 💀

3.3 Propositional Logic

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

Example:

a) It is Sunday.

b) The Sun rises from West (False proposition)

Following are some basic facts about propositional logic

Propositional logic is also called Boolean logic as it works on 0 and 1.

- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and logical connectives.
- These connectives are also called logical operators.
- 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.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "Where is Rohini", "How are you", "What is your name", are not propositions.

Syntax of propositional logic:

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

1) Atomic Propositions
2) Compound propositions

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

Example:

a) 2+2 is 4, it is an atomic proposition as it is a true fact.

b) "The Sun is cold" is also a proposition as it is a false fact.

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

Example:

a) "It is raining today, and street is wet."

b) "Ankit is a doctor, and his clinic is in Mumbai." Logical Connectives:

- Logical connectives are used to connect two simpler propositions or representing a sentence logically.

- We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

1) **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
2) **Conjunction:** A sentence which has ∧ connective such as, P ∧ Q is called a conjunction.

   Example: Rohan is intelligent and hardworking. It can be written as, P=

   Rohan is intelligent,

   Q= Rohan is hardworking. → P∧ Q.
3) **Disjunction:** A sentence which has ∨ connective, such as P ∨ Q. is called disjunction, where P and Q are the propositions.

   Example: "Ritika is a doctor or Engineer",

   Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as P ∨ Q.
4) **Implication:** A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as If it is raining, then the street is wet.

   Let P= It is raining, and Q= Street is wet, so it is represented as P → Q
5) **Biconditional:** A sentence such as P⇔ Q is a Biconditional sentence, example If I am breathing, then I am alive

P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

A formal grammar of propositional logic

Figure 7.7 gives a formal grammar of propositional logic; see page 984 if you are not familiar with the BNF notation.

Sentence	→	*AtomicSentence* | *ComplexSentence*
AtomicSentence	→	**True** | **False** | Symbol
Symbol	→	*P* | **Q** | **R** | ...
ComplexSentence	→	¬ *Sentence*
	|	(*Sentence* ∧ *Sentence*)
	|	(*Sentence* ∨ *Sentence*)
	|	(*Sentence* ⇒ *Sentence*)
	|	(*Sentence* ⇔ *Sentence*)

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic.

Propositional Logic: Semantics

- The semantics defines the rules for determining the truth of a sentence with respect to a particular problem (model).
- The semantic for propositional logic must specify how to compute the truth value of any sentence given a problem.

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

Truth Table:

For Negation:

P	¬ P
True	**False**
False	**True**

For Conjunction:

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

For disjunction:

P	Q	P ∨ Q.
True	True	**True**
False	True	**True**
True	False	**True**
False	False	**False**

For Implication:

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

For Biconditional:

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

Truth table with three propositions

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

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

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

Precedence	Operators
First Precedence	Parenthesis
Second Precedence	Negation
Third Precedence	Conjunction(AND)
Fourth Precedence	Disjunction(OR)
Fifth Precedence	Implication
Six Precedence	Biconditional

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

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

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

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

Properties of Operators:

Commutativity:

P∧ Q= Q ∧ P, or P ∨ Q = Q ∨ P.

Associativity:

(P ∧ Q) ∧ R= P ∧ (Q ∧ R), (P ∨ Q) ∨ R= P ∨ (Q ∨

R)

Identity element:

P ∧ True = P,

P ∨ True= True.

Distributive:

P∧ (Q ∨ R) = (P ∧ Q) ∨ (P ∧ R).

P ∨ (Q ∧ R) = (P ∨ Q) ∧ (P ∨ R).

DE Morgan's Law:

¬ (P ∧ Q) = (¬P) ∨ (¬Q)

¬ (P ∨ Q) = (¬ P) ∧ (¬Q).

Double-negation elimination:

¬ (¬P) = P.

A Simple Knowledge Base

- We have defined the semantics for propositional logic, we can construct a knowledge base for the Wumpus world
- Let Pi,j be true if there is a Pit in the room [i, j].
- Let Bi,j be true if agent perceives breeze in [i, j], (dead or alive).
- Let Wi,j be true if there is wumpus in the square[i, j].

The knowledge base includes the following sentences •

There is no pit in [1,1]:

$$R_1 : \quad \neg P_{1,1} .$$

- A square is breezy if and only if there is a pit in a neighbouring square. This has to be stated for each square; for now, we include just the relevant squares:

$$R_2 : \quad B_{1,1} \quad \Leftrightarrow \quad (P_{1,2} \vee P_{2,1}) .$$
$$R_3 : \quad B_{2,1} \quad \Leftrightarrow \quad (P_{1,1} \vee P_{2,2} \vee P_{3,1}) .$$

- Now we include the breeze percept for the first two squares visited in the specific world the agent is in

$$R_4 : \quad \neg B_{1,1} .$$
$$R_5 : \quad B_{2,1} .$$

- The knowledge base, then, consists of sentences R1 through R5.It can also be considered as a single sentence-the conjunction R1 A Rz A R3 A R4 A R5-because it asserts that all the individual sentences are true.

Inference

For propositional logic, models are assignments of true or false to every proposition symbol. Returning to our wumpus-world example, the relevant proposition symbols are $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$,$P_{2,1}$, $P_{2,2}$ and $P_{3,1}$.With seven symbols, there are $\mathbf{2^7}$=128 possible models; in three of these, **KB** is true.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 7.9 *A* truth table constructed for the knowledge base given in the text. KB is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows. In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

Propositional Theorem Proving:

Propositional Theorem Proving is a method used in Artificial Intelligence to prove whether a given statement (called a *theorem*) is true based on known facts (called *knowledge base* or *KB*).

It works only with propositional logic, meaning:

- Statements are either True or False.
- Statements are atomic (like P, Q, R) or combinations using AND (∧), OR (∨), NOT (¬), IMPLIES (→).

The goal is to prove that a query logically follows from the knowledge base.

Formula:

The basic idea:

$$\text{If } KB \models \alpha \quad \text{then} \quad KB \cup \{\neg\alpha\} \text{ is UNSATISFIABLE.}$$

Meaning:

- KB = Knowledge Base (facts you know)
- α = Query (the thing you want to prove)
- $\neg\alpha$ = Negation of the Query
- If KB + (NOT query) causes a **contradiction** (something impossible), then the query is TRUE.

✅ This is called **Proof by Contradiction**.

Step-by-Step Simple Example:

Suppose:

Knowledge Base (KB):

1. $P \rightarrow Q$ (If P then Q)
2. P (P is true)

Query:

- Prove Q is true.

Step 1: Write KB and Query

We have:

- KB $= \{ P \rightarrow Q, P \}$
- Query $= Q$

Step 2: Negate the Query

Negate Q:

- $\neg Q$

Now, the new KB is:

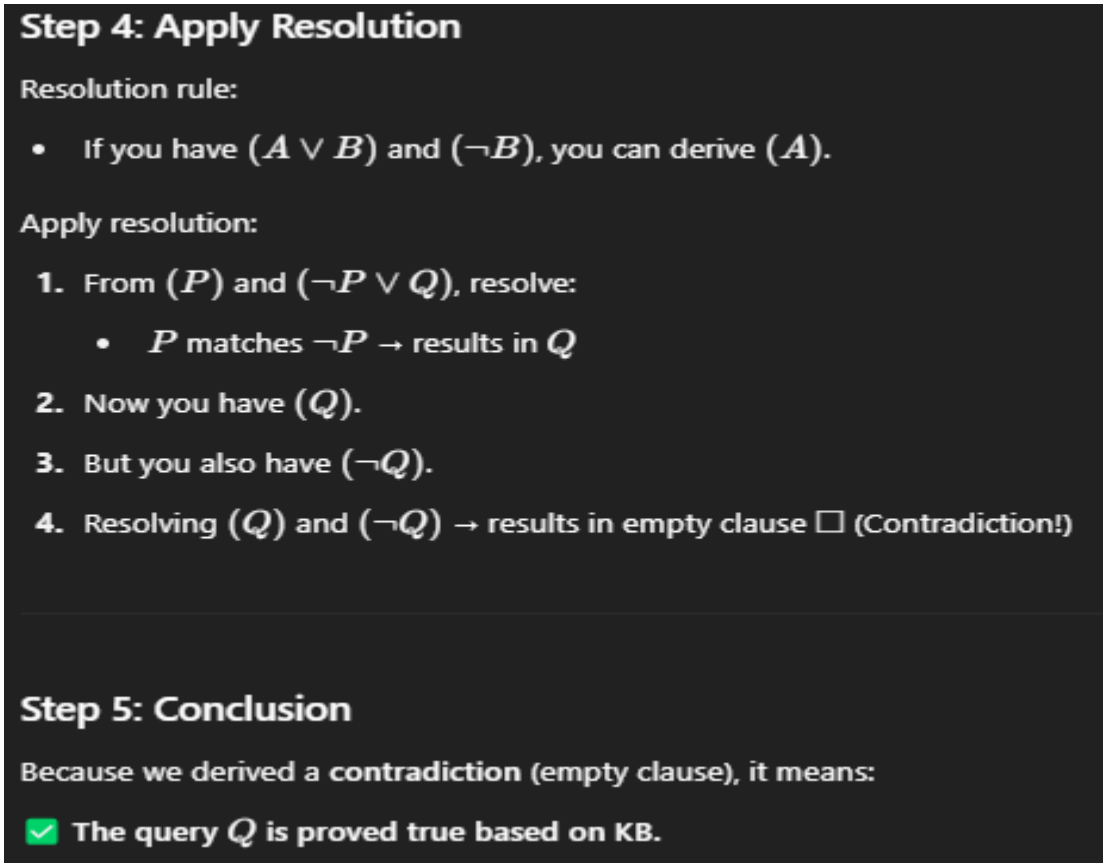
- $\{ P \rightarrow Q, P, \neg Q \}$

Step 3: Convert everything to CNF (Conjunctive Normal Form)

- $P \rightarrow Q$ is equivalent to $\neg P \vee Q$
- P stays as P
- $\neg Q$ stays as $\neg Q$

So CNF form is:

- $(\neg P \vee Q)$
- (P)
- $(\neg Q)$



Satisfiability: A formula is called satisfiable if it is true for at least one case.

Valid: A formula is called valid if it is true in all the cases or interpretation

Inference:

In artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, so generating the conclusions from evidence and facts is termed as Inference.

Inference rules:

Inference rules are the templates for generating valid arguments. Inference rules are applied to derive proofs in artificial intelligence, and the proof is a sequence of the conclusion that leads to the desired goal.

Notation for Modus ponens: $\frac{P \rightarrow Q, \quad P}{\therefore Q}$

$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$$ associativity of ∧
$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$$ associativity of ∨
$$\neg(\neg\alpha) \equiv \alpha$$ double-negation elimination
$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$$ contraposition
$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$$ implication elimination
$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$$ biconditional elimination
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$$ De Morgan
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$$ De Morgan
$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$$ distributivity of ∧ over ∨
$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$$ distributivity of ∨ over ∧

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

P	Q	P → Q	Q→ P	¬ Q → ¬ P	¬ P → ¬ Q.
T	T	T	T	T	T
T	F	F	T	F	T
F	T	T	F	T	F
F	F	T	T	T	T

Modus Ponens:

Example:

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q

Statement-2: "I am sleepy" ==> P

Conclusion: "I go to bed." ==> Q.

Hence, we can say that, if P→ Q is true and P is true then Q will be true.

Proof by Truth table:

P	Q	$P \rightarrow Q$
0	0	0
0	1	1
1	0	0
1	1	1

Modus Tollens:

Notation for Modus Tollens: $\frac{P \rightarrow Q, \quad \sim Q}{\sim P}$

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q

Statement-2: "I do not go to the bed."==> ~Q

Statement-3: Which infers that "I am not sleepy" => ~P

Proof by Truth table:

P	Q	~P	~Q	$P \rightarrow Q$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	0	0	1

And elimination:

$$(P \wedge Q) \rightarrow P$$

and

$$(P \wedge Q) \rightarrow Q$$

Agent Based on Propositional Logic

Example proof by deduction (Wumpus World): Let us take a condition which says there is breeze in [2,2] square and there is no breeze in [2,2] square.

3.4 Effective Propositional Model Checking

Model Checking is a major technique in Knowledge Representation and Reasoning (KRR), particularly for verifying if a knowledge base (KB) entails a given query. In propositional logic, effective model checking refers to systematically determining if a query is true in every model (interpretation) that satisfies the knowledge base.

- **Knowledgebase(KB):**A set of propositional logic sentences believed to be true.
- **Model:**An assignment of truth values (True or False) to each proposition.
- **Entailment(⊨):**KB ⊨ α means that in every model where KB is true, α is also true.
- **ModelChecking:**Checking all models of KB to see if α is true in all of them.

Effective Propositional Model Checking Workflow

1. **EnumerateModels:**
   List all possible truth assignments (there are 2^n models for n variables).
2. **FilterModels:**
   Keep only the models that make KB true.
3. **CheckQuery:**
   Confirm if in all these models, the query α is also true.

Example

KB:

- P → Q
- P

Query: Q

Process:

- Enumerate all models (P, Q combinations)
- Keep those where (P → Q) ∧ P is True
- Check if Q is True in all those models

Result:

- Only models where P is True and Q is True are valid.
- So Q must be true ⇒ KB ⊨ Q ✅

Aspect	Model Checking	Proof Checking
Approach	Checks truth across all models	Applies inference rules
Efficiency	Simple but potentially slower	More efficient in structured KBs
Strength	Good for small systems, verification tasks	Good for general reasoning

3.4.1complete bracktraking search(DPLL Algorithm):

Complete backtracking search is a systematic method for exploring all possible assignments to a set of variables in order to solve a constraint satisfaction problem, like SAT (Boolean satisfiability problem).

The DPLL algorithm (named after Davis, Putnam, Logemann, and Loveland) is a famous example of complete backtracking search applied specifically to SAT problems.

The DPLL algorithm improves basic backtracking by:

- Unit Propagation: If a clause has only one unassigned literal, that literal must be assigned to make the clause true.
- Pure Literal Elimination: If a variable always appears with the same polarity (always positive or always negative), assign it to satisfy all clauses it appears in.
- Early Termination: If all clauses are satisfied, we can immediately stop.
- Backtracking: If a contradiction (conflict) is found, the algorithm backtracks and tries a different assignment.

Example: Scheduling — say you're assigning courses to classrooms, and the constraints are:

- No two classes at the same time in the same room.
- Certain rooms can only host certain types of classes (like labs). You can encode this as a Boolean SAT problem and use DPLL to find a valid assignment of classes to rooms and times.

Formula Representation

DPLL works on formulas in Conjunctive Normal Form (CNF): A formula is a conjunction (AND, ∧) of clauses, where each clause is a disjunction (OR, ∨) of literals.

- A literal is a variable (e.g., AAA) or its negation (e.g., ¬A).
- Example of CNF formula**:**

$$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee \neg C)$$

In **model checking**, the idea is:

- The model (system) is expressed as a large propositional formula.
- Checking whether a property holds reduces to checking **whether a formula is satisfiable.**
- If SAT (satisfiable) → the property **might fail.**
- If UNSAT (unsatisfiable) → the property **is verified.**

Algorithm:

DPLL Algorithm (Complete Backtracking Search)

Input

- A formula Φ in CNF.
- An initial (possibly empty) partial assignment of truth values to variables.

Process

Step1: Simplify the formula using the current assignment:

- Remove satisfied clauses (where at least one literal is true).
- Remove falsified literals from remaining clauses.

Step2Check for Success or Failure:

- Success: If all clauses are satisfied (formula is empty), return SAT (a satisfying assignment was found).
- Failure: If any clause becomes empty (i.e., no literals left), return UNSAT (this path cannot work).

Step 3:Apply Unit Propagation:

- If any clause has only one unassigned literal (a unit clause), assign the necessary value to satisfy it.
- Repeat until no unit clauses are left.

Step 4:Apply Pure Literal Elimination:

- If a variable appears with only one polarity across all clauses (only XXX or only ¬X\neg X¬X), assign it to satisfy all such clauses.

Step 5:Choose a Variable (Decision Step):

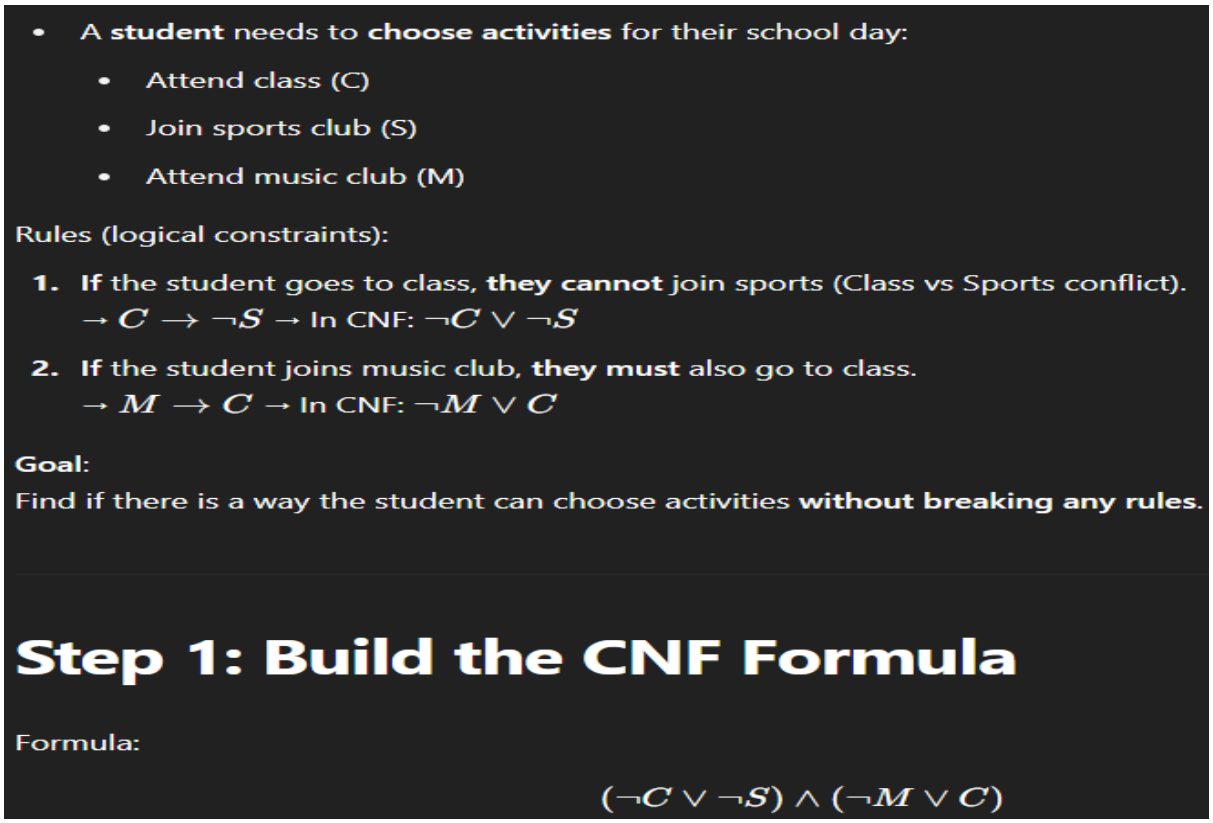
- Pick an unassigned variable.
- Assign it true (first guess).

Step 6:Recursive Search:

- Recursively call DPLL on the updated formula.
- If SAT is found, return SAT.

Step 7:Backtrack:

- If SAT not found (branch failed), flip the last decision (assign false) and retry.
- If both true and false assignments fail, backtrack further.



Simple Example program:

```
def dpll(clauses, assignment={}):
    clauses = [c for c in clauses if not any((assignment.get(x.strip('-'), x[0] != '-') for x in c))]
    if not clauses: return True, assignment
```

```
    if any(not c for c in clauses): return False, None
    unit = next((l for c in clauses if len(c)==1 for l in c), None)
    if unit:
        assignment[unit.strip('-')] = not unit.startswith('-')
        return dpll(clauses, assignment)
    var = next(l.strip('-') for c in clauses for l in c if l.strip('-') not in assignment)
    for val in [True, False]:
        new_assign = assignment.copy(); new_assign[var] = val
        sat, res = dpll(clauses, new_assign)
        if sat: return sat, res
    return False, None

clauses = [['-C', '-S'], ['-M', 'C']]
sat, res = dpll(clauses)
print("✅" if sat else "❌", res)
```

Output:

{'C': True, 'S': False, 'M': False}

3.4.2 Incomplete Local Search(Walksat Algorithm)

Incomplete Local Search refers to search methods that do not guarantee finding a complete solution even if one exists. They are typically faster and more practical for large problems, especially when completeness is less important than speed.

The WalkSAT algorithm is a famous incomplete local search algorithm for solving propositional satisfiability problems (SAT).

Here's how WalkSAT works:

- Start with a random assignment of truth values (true/false) to all variables.
- While the assignment does not satisfy the formula:
    - Randomly pick an unsatisfied clause (a disjunction of literals that is currently false).
    - With some probability (called the "walk probability"):
        - Randomly flip the truth value of a variable in that clause (even if it makes things worse).
    - Otherwise:
        - Flip the variable that would maximize the number of satisfied clauses (greedy move).

- Terminate if a satisfying assignment is found or after a certain number of steps (to avoid running forever).

It's "incomplete" because:

- It might get stuck in a local minimum.
- It might not find a satisfying assignment even if one exists.

Example:

Imagine you are **making a timetable** for classes (Math, English, Science) where:

- No two classes happen at the **same time**.
- A student **must attend** at least one science class a week.
- A teacher **cannot teach two classes at once**.

You can write these rules as a **logic formula** (in propositional logic).

👉 **Effective Propositional Model Checking** means:

- Use a **SAT solver** to **check** if your timetable **follows all rules**.
- The system **automatically finds** a timetable that works.
- If it can't, it shows you a **problem** (like double-booked teachers).

Student example in real life:

- University scheduling systems use **SAT-based model checking** to **build exam timetables**.
- They check millions of possibilities very **quickly** without conflicts.

Suppose you want to schedule:

- Math (M), English (E), Science (S)

Rules (turned into logic clauses):

- (M OR NOT E) ➔ Math and English can't be at the same time.
- (S) ➔ Must have Science.

WalkSAT would:

- Randomly pick times for M, E, S.
- See if any rule is broken.
- Flip one choice (randomly or wisely).
- Repeat until all rules are OK or give up.

Algorithm

Step 1: Start with a random guess

- **Give true or false values randomly to all variables.**

👉 ***(Example: Math = true, English = false, Science = true)***

Step 2: Check if all rules (clauses) are satisfied

- **If yes ➔ Done!** 🎉
- **If no ➔ Go to Step 3.**

Step 3: Pick a random broken rule (unsatisfied clause)

- **Choose one rule that is currently false.**

👉 ***(Example: "Math OR NOT English" is false if Math=false and English=true.)***

Step 4: Decide how to fix it

- **With some probability (example: 0.5):**
  - **Randomly flip the value of one variable in that rule.**
- **Otherwise:**
  - **Flip the variable that makes the most clauses true.**

👉 ***(Example: Flip Math from false ➔ true.)***

Step 5: Repeat

- **Go back to Step 2.**
- **Keep repeating until:**
  - **You satisfy all the rules, or**
  - **You reach a maximum number of tries and give up.**

Simple Program:

```
import random
clauses = [[1, 2], [-1, -2]]  # (A or B) and (not A or not B)
assignment = {1: random.choice([True, False]), 2: random.choice([True, False])}

for _ in range(10):
    if all(any(assignment[abs(lit)] if lit > 0 else not assignment[abs(lit)] for lit in clause) for clause in clauses):
        break
    clause = random.choice(clauses)
    var = abs(random.choice(clause))
    assignment[var] = not assignment[var]
```

print(f'Student A attends: {assignment[1]}, Student B attends: {assignment[2]}")

Output:

Student A attends: True, Student B attends: False

Feature	DPLL (Complete Search)	WalkSAT (Incomplete Search)
Type	**Complete (Backtracking search)**	**Incomplete (Local search)**
Guarantee	**Always finds a solution if one exists (or proves unsatisfiable)**	**Might find a solution, but not guaranteed**
Method	**Systematically explores all possibilities (splitting and backtracking)**	**Randomly flips variables to fix broken clauses**
Goal	**Prove SAT or UNSAT (full exploration if needed)**	**Quickly find a satisfying assignment (if lucky)**
Example Strategy	**Choose a variable → Assign true/false → Recurse → Backtrack if needed**	**Pick an unsatisfied clause → Flip a random variable inside**
Speed	**Slow on large problems (but accurate)**	**Fast on large problems (but may fail)**
Memory Usage	**Higher (due to recursion and backtracking)**	**Lower (just maintain current assignment)**
Typical Use	**When correctness and proof are required**	**When quick approximate solution is enough**

3.5 Agents Based on Propositional Logic

1. The current state of the world

We can associate proposition with timestamp to avoid contradiction.

e.g. $\neg Stench^3$, $Stench^4$

fluent: refer an aspect of the world that changes. (E.g. $L^t_{x,y}$)

atemporal variables: Symbols associated with permanent aspects of the world do not need a time superscript.

Effect axioms: specify the outcome of an action at the next time step.

Frame problem: some information lost because the effect axioms fails to state what remains

unchanged as the result of an action.

Solution: add frame axioms explicity asserting all the propositions that remain the same.

Representation frame problem: The proliferation of frame axioms is inefficient, the set of frame axioms will be O(mn) in a world with m different actions and n fluents.

Solution: because the world exhibits **locaility** (for humans each action typically changes no more than some number k of those fluents.) Define the transition model with a set of axioms of size O(mk) rather than size O(mn).

Inferential frame problem: The problem of projecting forward the results of a t step lan of action in time O(kt) rather than O(nt).

Solution: change one's focus from writing axioms about actions to writing axioms about fluents.

For each fluent F, we will have an axiom that defines the truth value of F^{t+1} in terms of fluents at time t and the action that may have occurred at time t.

The truth value of F^{t+1} can be set in one of 2 ways:

Either a. The action at time t cause F to be true at t+1

Or b. F was already true at time t and the action at time t does not cause it to be false.

An axiom of this form is called a **successor-state axiom** and has this schema:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t) .$$

Qualification problem: specifying all unusual exceptions that could cause the action to fail.

The Current State of the World

- **Timestamped propositions** help avoid contradictions when things change (e.g., $\neg Stench_3$, $Stench_4$).
- **Fluents** represent aspects of the world that change over time (e.g., $L_{x,y}$).
- **Atemporal variables** stay the same across time (no timestamp needed).
- **Effect axioms** describe what changes after an action.

Frame Problem:

- Issue: Effect axioms don't say what stays the same, leading to loss of information.
- Solution: Add **frame axioms** to explicitly state unchanged propositions.

Representation Frame Problem:

- Issue: Too many frame axioms (O(mn)) if every action and fluent is considered.
- Solution: Assume **locality**: each action only affects k fluents → model size O(mk).

Inferential Frame Problem:

- Issue: Predicting future states can be too slow (O(nt) time).
- Solution: Use **successor-state axioms**:
    - **Ft+1** is true if:
        - (a) An action at t causes F to be true at t+1, OR
        - (b) F was true at t and the action doesn't make it false.

Qualification Problem:

- It's hard to list all rare exceptions that could make an action fail.

2. A hybrid agent

Hybrid agent: combines the ability to deduce various aspect of the state of the world with condition-action rules, and with problem-solving algorithms.

The agent maintains and update KB as a current plan.

The initial KB contains the atemporal axioms. (don't depend on t)

At each time step, the new percept sentence is added along with all the axioms that depend on t (such as the successor-state axioms).

Then the agent use logical inference by ASKING questions of the KB (to work out which squares are safe and which have yet to be visited).

The main body of the agent program constructs a plan based on a decreasing priority of goals:

1. If there is a glitter, construct a plan to grab the gold, follow a route back to the initial location and climb out of the cave;
2. Otherwise if there is no current plan, plan a route (with A* search) to the closest safe square unvisited yet, making sure the route goes through only safe squares;
3. If there are no safe squares to explore, if still has an arrow, try to make a safe square by shooting at one of the possible wumpus locations.
4. If this fails, look for a square to explore that is not provably unsafe.
5. If there is no such square, the mission is impossible, then retreat to the initial location and climb out of the cave.

- Combines **logical inference**, **condition-action rules**, and **problem-solving**.
- Maintains a **Knowledge Base (KB)** updated each time step.
- **Atemporal axioms** stay fixed; **time-dependent axioms** are updated.
- Uses **ASK** operations to infer safe moves.

Agent Planning Strategy (Priority Order):

1. If glitter detected → grab gold → return and climb out.
2. Else, if no plan → plan to explore nearest **safe unvisited** square (A* search).

3. Else, if possible → **shoot** at possible Wumpus locations.
4. Else, explore squares not **provably unsafe**.
5. Else → **give up** and return to initial location.

Weakness:

- **Computation cost** increases over time as KB grows.

function HYBRID-WUMPUS-AGENT(*percept*) **returns** an *action*
  inputs: *percept*, a list, [*stench*,*breeze*,*glitter*,*bump*,*scream*]
  persistent: *KB*, a knowledge base, initially the atemporal "wumpus physics"
    t, a counter, initially 0, indicating time
    plan, an action sequence, initially empty

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  TELL the *KB* the temporal "physics" sentences for time *t*
  safe ← {[*x*, *y*] : ASK(*KB*, $OK^t_{x,y}$) = *true*}
  if ASK(*KB*, $Glitter^t$) = *true* **then**
    plan ← [*Grab*] + PLAN-ROUTE(*current*, {[1,1]}, *safe*) + [*Climb*]
  if *plan* is empty **then**
    unvisited ← {[*x*, *y*] : ASK(*KB*, $L^{t'}_{x,y}$) = *false* for all $t' \leq t$}
    plan ← PLAN-ROUTE(*current*, *unvisited* ∩ *safe*, *safe*)
  if *plan* is empty and ASK(*KB*, $HaveArrow^t$) = *true* **then**
    possible_wumpus ← {[*x*, *y*] : ASK(*KB*, ¬ $W_{x,y}$) = *false*}
    plan ← PLAN-SHOT(*current*, *possible_wumpus*, *safe*)
  if *plan* is empty **then** // no choice but to take a risk
    not_unsafe ← {[*x*, *y*] : ASK(*KB*, ¬ $OK^t_{x,y}$) = *false*}
    plan ← PLAN-ROUTE(*current*, *unvisited* ∩ *not_unsafe*, *safe*)
  if *plan* is empty **then**
    plan ← PLAN-ROUTE(*current*, {[1, 1]}, *safe*) + [*Climb*]
  action ← POP(*plan*)
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  t ← *t* + 1
  return *action*

function PLAN-ROUTE(*current*,*goals*,*allowed*) **returns** an action sequence
  inputs: *current*, the agent's current position
    goals, a set of squares; try to plan a route to one of them
    allowed, a set of squares that can form part of the route

  problem ← ROUTE-PROBLEM(*current*, *goals*,*allowed*)
  return A*-GRAPH-SEARCH(*problem*)

Figure 7.20 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to decide what actions to take.

Weakness: The computational expense goes up as time goes by.

Step1: Get the new percept and update the knowledge base (KB) about time t.

Step 2:Mark squares as "safe" if KB says they are safe.

Step 3:If there's glitter, plan to grab gold and escape.

Step 4:If no plan, find a safe unvisited square and plan to go there.

Step 5:If still no plan and you have an arrow, find possible Wumpus and plan to shoot.

Step 6:If still no plan, take a risk: find a not-proven-unsafe square.

Step 7:If still no plan, retreat back to start and climb out.

Step 8:Pop the next action from the plan.

Step 9:Update the KB with the action taken and move to next time step.

Step 10:Return the chosen action to execute.

3. Logical state estimation

To get a constant update time, we need to **cache** the result of inference.

Belief state: Some representation of the set of all possible current state of the world. (used to replace the past history of percepts and all their ramifications)

e.g.

$$WumpusAlive^1 \wedge L^1_{2,1} \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2})$$

We use a logical sentence involving the proposition symbols associated with the current time step and the temporal symbols.

Logical **state estimation** involves maintaining a logical sentence that describes the set of possible states consistent with the observation history. Each update step requires inference using the transition model of the environment, which is built from **successor-state axioms** that specify how each **fluent** changes.

State estimation: The process of updating the belief state as new percepts arrive.

Exact state estimation may require logical formulas whose size is exponential in the number of symbols.

One common scheme for approximate state estimation: to represent belief state as conjunctions of literals (1-CNF formulas).

The agent simply tries to prove X^t and $\neg X^t$ for each symbol X^t, given the belief state at t-1.

The conjunction of provable literals becomes the new belief state, and the previous belief state is discarded.

(This scheme may lose some information as time goes along.)

The set of possible states represented by the 1-CNF belief state includes all states that are in fact possible given the full percept history. The 1-CNF belief state acts as a simple outer envelope, or **conservative approximation**.

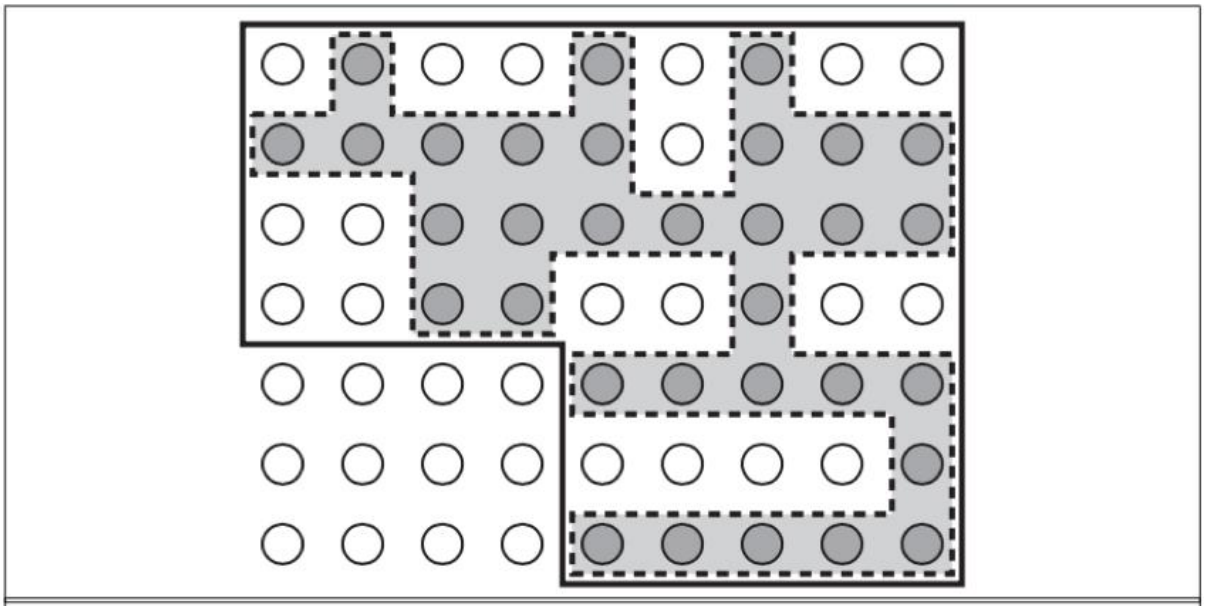


Figure 7.21 Depiction of a 1-CNF belief state (bold outline) as a simply representable, conservative approximation to the exact (wiggly) belief state (shaded region with dashed outline). Each possible world is shown as a circle; the shaded ones are consistent with all the percepts.

- **Goal**: Constant-time updates by caching inference results.
- **Belief state**: Logical representation of all possible current world states.
- Updates using **successor-state axioms** and percepts.

Exact state estimation:

- Can become **exponentially large**.

Approximate state estimation:

- Use **1-CNF belief states** (simple conjunctions of literals).
- Only track what can be **proven** true or false.

Tradeoff:

- Approximation may lose information but stays **conservative** (never rules out possible worlds).

4. Making plans by propositional inference

We can make plans by logical inference instead of A* search in Figure 7.20.

Basic idea:

1. Construct a sentence that includes:

a) $Init^0$: a collection of assertions about the initial state;

b) $Transition^1$, …, $Transition^t$: The successor-state axioms for all possible actions at each time

up to some maximum time t;

c) $HaveGold^t \wedge ClimbedOut^t$: The assertion that the goal is achieved at time t.

2. Present the whole sentence to a SAT solver. If the solver finds a satisfying model, the goal is achievable; else the planning is impossible.

3. Assuming a model is found, extract from the model those variables that represent actions and are assigned true.

Together they represent a plan to ahieve the goals.

Decisions within a logical agent can be made by SAT solving: finding possible models specifying future action sequences that reach the goal. This approach works only for fully observable or sensorless environment.

SATPLAN: A propositional planning. (Cannot be used in a partially observable environment)

SATPLAN finds models for a sentence containing the initial sate, the goal, the successor-state axioms, and the action exclusion axioms.

(Because the agent does not know how many steps it will take to reach the goal, the algorithm tries each possible number of steps t up to some maximum conceivable plan length T_{max}.)

```
function SATPLAN(init, transition, goal, T_max) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
          T_max, an upper limit for plan length

  for t = 0 to T_max do
    cnf ← TRANSLATE-TO-SAT(init, transition, goal, t)
    model ← SAT-SOLVER(cnf)
    if model is not null then
      return EXTRACT-SOLUTION(model)
  return failure
```

Figure 7.22 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step t and axioms are included for each time step up to t. If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned *true* in the model. If no model exists, then the process is repeated with the goal moved one step later.

Precondition axioms: stating that an action occurrence requires the preconditions to be satisfied, added to avoid generating plans with illegal actions.

Action exclusion axioms: added to avoid the creation of plans with multiple simultaneous actions that interfere with each other.

First-Order Logic-Syntax and Semantics of First-Order Logic

First-Order Logic (FOL), also called **Predicate Logic**, is a formal system used in **Artificial Intelligence** to describe the **relationships between objects** and their **properties**. It extends **propositional logic** by adding **quantifiers** ("for all", "there exists") and **variables** that can represent objects in the world.

In simple words:

FOL lets AI reason about "things" (like people, cars, etc.), what properties they have (like being red, smart), and how they are related to each other (like "owns", "loves").

Syntax of First-Order Logic (How FOL is written)

- **Constants**: Specific objects (e.g., `John`, `Paris`, `5`).
- **Variables**: Placeholders for objects (e.g., `x`, `y`, `z`).
- **Predicates**: Properties or relations (e.g., `Loves(John, x)`, meaning John loves x).
- **Functions**: Mappings from objects to objects (e.g., `Mother(x)` gives the mother of x).
- **Quantifiers**:
  - Universal (`∀`): "for all"
    Example: ∀x Human(x) → Mortal(x)
    ("All humans are mortal")
  - Existential (`∃`): "there exists"
    Example: ∃x Loves(x, Mary)
    ("Someone loves Mary")
- **Connectives**:
  - ∧ (and), ∨ (or), ¬ (not), → (implies), ↔ (if and only if)

Semantics of First-Order Logic (What FOL *means*)

Semantics defines **how we interpret** the syntax:

- Each **constant** points to a specific object.
- Each **predicate** refers to a property or a relationship among objects.
- **Quantifiers** determine how many objects the statement refers to.

In short:

Semantics gives *meaning* to the sentences we write in FOL, connecting them to the real world.

ExampleSituation:

Suppose we're building an AI for an online shopping system.

We want to represent:

- "Every customer who buys a product likes it."

In First-Order Logic:

∀x ∀y (Customer(x) ∧ Product(y) ∧ Buys(x, y) → Likes(x, y)) **Meaning:** For every customer x and every product y, if x buys y, then x likes y.

Another Simple Real-World Example

- **Statement:** "There is a doctor who treats every patient."
- **FOL Representation:**

∃d (Doctor(d) ∧ ∀p (Patient(p) → Treats(d, p)))

Meaning:

There exists some d who is a doctor, and for all p, if p is a patient, then d treats p.

Difference Between syntax first order logic and semantic of first order logic.

Aspect	Syntax	Semantics
Meaning	Syntax is about the *structure* and *rules* for building valid expressions or formulas.	Semantics is about the *meaning* or *interpretation* of those formulas.
Focus	Focuses on *how* formulas are written (symbols, grammar).	Focuses on *what* formulas *mean* in a particular domain.
Example	Rules for writing: ∀x (P(x) → Q(x)) is a syntactically correct formula.	Interpretation: ∀x (P(x) → Q(x)) might mean "For every person x, if x is a student (P(x)), then x studies (Q(x))."
Key elements	Variables, constants, functions, predicates, logical connectives (¬, ∧, ∨, →), quantifiers (∀, ∃).	Domains of discourse (set of things we are talking about), interpretation of functions, predicates, and constants.
Concerned with	Whether a statement is *well-formed* (correctly written).	Whether a statement is *true* or *false* under a certain interpretation.

3.8 Using First-Order Logic

- In propositional logic, we can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power.

First-Order logic:

- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.

- FOL is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as Predicate logic or First-order predicate logic. Firstorder logic is a powerful language that develops information about the objects in an easier way and can also express the relationship between those objects.
- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:

Objects: A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, ...

Relations: It can be unary relation such as: red, round, is adjacent, or n-any relation such as: the sister of, brother of, has color, comes between

Function: Father of, best friend, third inning of, end of, ...

- As a natural language, first-order logic also has two main parts:
  1) Syntax
  2) 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.

Basic Elements of First-order logic:

Following are the basic elements of FOL syntax:

Constant	**1, 2, A, John, Mumbai, cat,....**
Variables	**x, y, z, a, b,....**
Predicates	**Brother, Father, >,....**
Function	**sqrt, LeftLegOf, ...**
Connectives	**∧, ∨, ¬, ⇒, ⇔**
Equality	**==**
Quantifier	**∀, ∃**

Example: "Evil King John ruled England in 1200."

Objects: John, England, 1200; Relation: ruled; Properties: evil, king.

Models for first-order logic

A model with five objects: Richard the Lionheart, King of England from 1189 1:o1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

The objects in the model may be related in various ways. In the figure, Richard and John are brothers.

Thus, the brotherhood relation in this model is the set { (Richard the Lionheart, King John), (King John, Richard the Lionheart)).

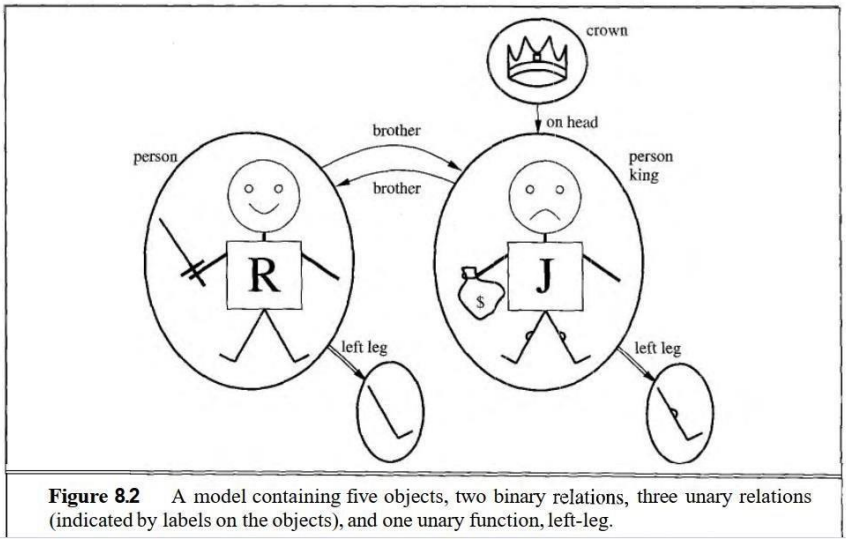


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

Objects: person King John

Person Richard

Crown

Relation: "on head" <the crown, king john>

"brother" <john, Richard>

Function: <John the king> -on-head (crown)

<John the king> - shoe(left-leg)

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).

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.

Example: >(1,2)V≤(1,2) => True

Sisters(geeta,seta)^sisters(geeta,leela)

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.

There are two types of quantifier:

1. Universal Quantifier, (for all, everyone, everything)
2. Existential quantifier, (for some, at least one).

Universal Quantifier:

- Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.
- The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.
- If x is a variable, then ∀x is read as:
  - **For all x**
  - **For each x**
  - **For every x.**

Example: "All cat like fish"

Let us take a variable x which can take the value of "cat". Let us take a predicate cat(x) which is true if x is a cat. Similarly let us take another predicate likes(x,y) which is true if x likes y.

∀(x) cat(x) =>likes(x,y) [y is fish] For

all x, if x is cat, then x likes y

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 ∃. 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 student like ice cream"

Let us take a variable x which can take the value of "student". Let us take a predicate student(x)

which is true if x is a student. Similarly let us take another predicate likes(x,y) which is true if x likes y.

∃(x) student(x)∧likes(x,y) [y is ice cream] There exists some x such that

x is a student and also likes ice cream.

Properties of Quantifiers:

1.In universal quantifier,

∀x∀y is similar to ∀y∀x.

2. In Existential quantifier,

∃x∃y is similar to ∃y∃x.

3.∃x∀y is not similar to

∀y∃x.

Equality

We can use the **equality symbol** to make statements to the effect that two terms refer to the same object. For example, Father(John) = Henry says that the object referred to by Father(John) and the object referred to by *Henry* are the same.

Using FOL

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions.** For example, we can assert that John is a king and that kings are persons:

$$\text{TELL}(KB, King(John)).$$
$$\text{TELL}(KB, \forall x \ King(x) \Rightarrow Person(x)) .$$

We can ask questions of the knowledge base using ASK. For example,

$\text{ASK}(KB, King(John))$ should also return *true.*

3.9Unification and Lifting

- 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 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.

UNIFY (P, Q) = □ where SUBST (□ , P) = SUBST(□ , Q)

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.

- 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].

Algorithm

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

Step.2: Recursively unify atomic sentences:

i. Check for Identical expression match. ii. If one expression is a variable vi, and the other is a term ti which does not contain variable vi, then:

I. Substitute ti / vi in the existing substitutions II. Add ti /vi to the substitution setlist.

III. 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.

Example:

Given: Knows(Ram, X) is a predicate **Whom does Ram knows?**

The UNIFY algorithm will search all the related sentences in the knowledge base, which could unify with Knows(Ram, X)

- UNIFY(Knows(Ram, X), Knows(Ram, Shyam)) = {X/Shyam} Here X is substituted with Shyam
- UNIFY (Knows(Ram, X), Knows(Y, Akash)) = {X/Akash, Y/Ram}
- Here X is substituted with Akash and Y is substituted with Ram From the unification it is inferred that Ram knows Shyam and Akash **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.

Example: UNIFY Knows (Ram,X) with Brother (Laxman, Ram)
Fails because predicate is different • Number of Arguments in both expressions must be identical.

Example: UNIFY hate (Marcus) with hate (Marcus, john)

Fails because number of arguments are different • Unification will fail if there are

two similar variables present in the same expression. Example: UNIFY Knows (Ram, X) with Knows (X, Raman)

Inference Engine

- The inference engine is the component of the expert system in AI, which applies logical rules to the knowledge base to infer new information from known facts.
- Forward and Backward chaining is the strategies used by the inference engine in making the deductions.

Example of Inference chain

Rule 1: IF X is true
THEN A is true

Rule 2: IF A is true
AND B is true
AND C is true
THEN Y is true

Rule 3: IF Y is true
AND D is true
THEN Z is true

Aspect	Unification	Lifting
Goal	Make two logical expressions identical by finding a substitution	Move reasoning from ground instances to general, variable-based expressions
Nature	Substitution process	Generalization process
Example	Matching likes(X, ice_cream) with likes(john, Y)	Working with likes(X, Y) instead of specific people and things
Used in	Resolution, matching, inference steps	Lifted resolution, abstract reasoning

Example program:

```python
# Knowledge Base
students = ["john", "mary", "alice"]

# Rule: If X is a student, then X is eligible for discount
def is_student(name):
    return name.lower() in students

def eligible_for_discount(name):
    # Unification step: Try to match name with a known student
    if is_student(name):
        print(f"Yes, {name.capitalize()} is eligible for a student discount.")
    else:
        print(f"No, {name.capitalize()} is not eligible for a student discount.")

# Test Queries
eligible_for_discount("Mary")
eligible_for_discount("Bob")
```

Output:

Yes, rajendra is eligible for a
student discount.

No, sandya is not eligible for a
student discount.

Forward Chaining, Backward Chaining.

Forward Chaining, Backward Chaining.

Forward chaining and Backward Chaining

Forward Chaining:

- It's a reasoning method that starts with the **available facts** and applies rules to infer new facts until a **goal or conclusion** is reached.
- It's **data-driven**: you move forward from known information to discover new information.
- Common in **expert systems** and **automation**
- The Forward Chaining Algorithm in AI is a data-driven reasoning approach that starts with known facts and applies inference rules to derive new facts until a desired goal is reached.
- Begin with **initial facts** in a knowledge base.
- Apply **rules** that match these facts to infer new facts.
- Continue this process until the **goal state** is achieved or no more new facts can be inferred.
- It start from initial state.

Example:

"If I know it's raining and I have an umbrella, I can infer that I'll stay dry."

- 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.

Properties of Forward-Chaining:

- 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 data-driven as we reach to the goal using available data.

Example:

Rule 1: IF student is excellent in academics

AND he is good in sports

THEN he is all rounder (Goal)

Rule 2: IF student gets marks > 80

AND he is good in genera knowledge

THEN he is excellent in academics

Rule 3: IF student is making centuries

THEN he is good in sports

Given following facts:

1. Student gets marks > 80
2. Student is making centuries
3. He is good in general knowledge **Explanation:**

- In forward chaining we need to check the given facts in the IF statement, the first given fact is present in the Rule 2 (i.e **Student gets marks > 80**).
- In the Rule 2 along with the first given fact, third given fact is also present (i.e, **He is good in general knowledge**).
- From the Rule 2, it is inferred that **he is excellent in academics** (the subgoal).
- Now remaining fact should be checked (i.e, **student is making centuries**)
- This fact is present in the Rule 3. From that it is inferred that **he is good in sports** (subgoal).

- The two inferred subgoals are given in the Rule 1 IF, AND statement.
- From this it is inferred that **he is all rounder**, which is the final goal.

Algorithm:

Step 1: Initialize

- Start with known facts.
- Define rules in the form IF (conditions) THEN (conclusion).
- Set a goal fact to find.

Step 2: Find Matching Rules

- Check which rules apply based on the current facts.

Step 3: Infer New Facts

- If a rule's conditions are met, add the conclusion as a new fact.
- Repeat this for new facts.

Step 4: Continue Until Goal is Found

- Keep applying rules until:
  - Goal is reached, OR
  - No more facts can be added.

Step 5: Show Result

- If the goal is found, print "Goal achieved!".
- Otherwise, print "Goal not reachable.".

Source Code:

Example Program:

```python
import tkinter as tk
from tkinter import messagebox
def forward_chaining(marks):
    if marks >= 90:
        return "A+"
    elif marks >= 80:
        return "A"
    elif marks >= 70:
        return "B+"
    elif marks >= 60:
        return "B"
    elif marks >= 50:
        return "C+"
    else:
        return "C"
def evaluate():
    try:
        marks = int(marksentry.get())
        grade = forward_chaining(marks)
        resultlabel.config(text=f"Grade: {grade}")
    except ValueError:
        messagebox.showerror("Invalid Input", "Please enter a valid number")
# Set up the main window
root = tk.Tk()
root.title("Student Marks Evaluation")
# Create widgets
markslabel= tk.Label(root, text="Enter Marks:")
markslabel.pack()
marksentry = tk.Entry(root)
marksentry.pack()
evaluatebutton = tk.Button(root, text="Evaluate", command=evaluate)
evaluatebutton.pack()
resultlabel = tk.Label(root, text="Grade: ")
resultlabel.pack()
# Start the GUI
root.mainloop()
```

Output: Input:

Output:

Backward Chaining

- It's a reasoning method that starts with a **goal** and works **backward** to find the facts that support that goal.
- It's **goal-driven**: you try to prove or disprove a hypothesis by finding supporting facts.
- Common in **diagnosis systems** and **problem-solving AI**.
- Backward chaining is an inference method used in artificial intelligence, particularly in rule-based expert systems.
- It starts with a goal (hypothesis) and works backward to find supporting facts or rules.
- It started in Goal state.
- Starts with a goal and works backward

Example:
"If I want to know if I'll stay dry, I check: Do I have an umbrella? Is it raining?"

A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

Properties of backward chaining:

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.

Example:

Rule 1: IF student is excellent in academics

AND he is good in sports

THEN he is all rounder (Goal)

Rule 2: IF student gets marks > 80

AND he is good in genera knowledge

THEN he is excellent in academics

Rule 3: IF student is making centuries.

THEN he is good in sports **Given following facts:**

1. Student gets marks > 80
2. Student is making centuries
3. He is good in general knowledge

Explanation:

- In Backward chaining from the goal we need to reach the given facts. Here the goal is he is all rounder**.**
- First we need to check whether goal is present in given facts, if not the statements in IF, AND condition should be treated as subgoals. In the given example IF **student is excellent in academics** and AND **he is good in sports are the subgoals**.
- Next check the subgoals in IF statement, **student is excellent in academics** is present in Rule 2 inferred part, so from that we got two given facts i.e **student gets marks > 80 and he is good in genera knowledge**.
- Next remaining subgoal (i.e **he is good in sports**) is checked in inferred part; it is present in Rule 3. From this we got one more fact i.e, **student is making centuries**.
- Finally we got all the given facts from the inferred statements.

Algorithm:

Step1: Start with the Goal

- Identify what you want to prove (goal).
- Add it to a list called GOALS (like a to-do list).

Step2: Check if the Goal is Already Known

- If the goal is already a fact in the knowledge base (KB), then it is TRUE (proven).

Step3: Look for Rules to Support the Goal

- Find rules that can help prove the goal.
- If no rule supports it, return FALSE (cannot be proven).

Step4: Check the Rule's Conditions

- Each rule has conditions that must be true.
- List all these conditions (sub-goals).

Step5: Prove Each Condition One by One

- If a condition is already a fact, mark it as TRUE.
- If not, repeat the backward chaining process for that condition.

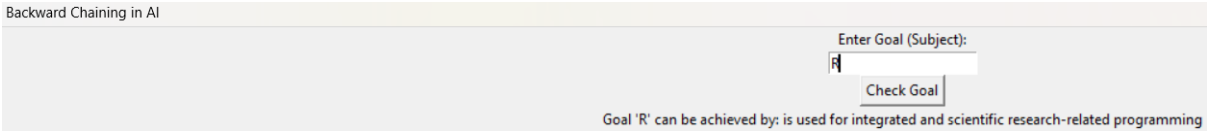
Step6: Decide the Final Result

- If at least one rule proves the goal, return TRUE.
- If no rule works, return FALSE.

Example Program:

```
import tkinter as tk
class BackwardChaining:
    def __init__(self):
        # Knowledge base as rules (goal -> fact)
        self.rules = {
            1: ("AI", "has knowledge of algorithms"),
            2: ("C", "is the basic of all programming languages"),
            3: ("JAVA", "is a trending language"),
            4: ("R", "is used for integrated and scientific research-related programming")
        }
        self.facts = []
    def backward_chaining(self, goal):
        self.facts.clear()  # Clear previous facts
        for rule in self.rules.values():
            if rule[0] == goal: # Case-insensitive comparison
                self.facts.append(rule[1])
    def display_result(self, goal):
        self.backward_chaining(goal)
        if self.facts:
            return f"Goal '{goal}' can be achieved by: {', '.join(self.facts)}"
        return f"Goal '{goal}' cannot be achieved."
def run_program():
    goal = entry.get()
    bc = BackwardChaining()
    result = bc.display_result(goal)
    resultlabel.config(text=result)
root = tk.Tk()
root.title("Backward Chaining in AI")
label = tk.Label(root, text="Enter Goal (Subject):")
label.pack()
entry = tk.Entry(root)
entry.pack()
submitbtn = tk.Button(root, text="Check Goal", command=run_program)
submitbtn.pack()
resultlabel = tk.Label(root, text="")
resultlabel.pack()
root.mainloop()
```

Output:



Aspect	Forward Chaining	Backward Chaining
Direction	Starts from **known facts** and moves **toward a conclusion**.	Starts from a **goal (hypothesis)** and works **backward to find facts** supporting it.
Driven by	**Data-driven** (what we know).	**Goal-driven** (what we want to prove).
Use case	When you have a lot of facts and want to discover what can be	When you have a goal in mind and want to check if it can be achieved based on

	concluded.	available facts.
Example	Diagnosis system gathering symptoms to conclude the disease.	Detective trying to verify if a suspect committed a crime.
Efficiency	May explore many possibilities.	More focused; only explores paths relevant to the goal.