UNIT – 3

TOPICS

- *Logical Agents*
  - Knowledge Based Agents
  - Logic
  - Propositional logic
  - First order logic
  - Syntax and Semantics in First order Logic
- *Inference in first order logic*
  - propositional vs. First order inference
  - Unification and Lifting
  - Forward chaining, Backward chaining
  - Resolution

Knowledge Based Agents

- An intelligent agent needs **knowledge** about the real world for taking decisions and **reasoning** to act efficiently.
- Knowledge-based agents are those agents who have the capability of ***maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently***.
- Knowledge based agents give the current situation in the form of sentences.
  - They have complete knowledge of current situation of mini-world and its surroundings.
  - These agents manipulate knowledge to infer new things at "Knowledge level".

Knowledge-Based Agents

- **Representation and Reasoning:**

1. Agents Form Internal Representations:

- Imagine an agent in a smart home setting. The agent represents the home's layout, temperature in each room, locations of inhabitants, and status of devices (lights, thermostat, etc.).

2. Inference to Derive New Representations and Actions:

- The agent can infer new knowledge from existing data. For instance, if it knows that the living room light is off and someone is watching TV there, it can infer that they might want more light and turn on the living room lamp.

Human Intelligence

- **Reasoning with Internal Knowledge Representations:**
- **Reasoning Example**:
  - A person sees that it's cloudy and deduces it might rain. They bring an umbrella before leaving the house. This decision is based on the internal representation of knowledge about clouds and rain.

AI Intelligence

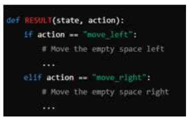
- **Mirrored in Knowledge-Based Agents:**
- **Smart Home Agent Example**:
  - The smart home agent has rules and knowledge encoded about typical human behaviors and preferences.
  - It observes that the temperature outside is dropping.
  - Given the internal representation that the inhabitants prefer the living room to be warm in the evening, the agent turns on the heater in the living room.

Problem-Solving Agents

- **Limitations**:
  - Previous problem-solving agents had limited and inflexible knowledge representations.
  - Example: The transition model in the 8-puzzle is hidden in the code, limiting its deductive capabilities.
- **Atomic Representations**:
  - In partially observable environments, listing all possible concrete states is impractical.

Limitations of Hidden Transition Model

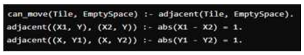
- **Lack of Explicit Knowledge**:
  - ◦ The knowledge of how actions affect the puzzle's state is not represented explicitly. The agent just follows pre-defined rules without understanding underlying principles.
- **No General Reasoning**:
  - ◦ The agent can execute moves but cannot reason about broader concepts. For example, it cannot deduce that two tiles cannot occupy the same space or infer properties about states (e.g., parity).
- **Inflexibility**:
  - ◦ This approach is domain-specific. The transition model is tightly coupled with the puzzle's specific rules, making it hard to apply the same logic to other problems or reason in a more abstract manner.



```
def RESULT(state, action):
    if action == "move_left":
        # Move the empty space left
        ...
    elif action == "move_right":
        # Move the empty space right
        ...
```

Comparison with Knowledge-Based Agents

- **Explicit Representation**:
  - ◦ A knowledge-based agent would explicitly represent the transition model using logical statements or rules. For instance, it might have a rule that states, "If tile X is adjacent to the empty space, then tile X can move to the empty space."
- **Deductive Capabilities**:
  - ◦ Such an agent could deduce more general principles. For example, it could infer constraints like "No two tiles can occupy the same space" or "If a tile is in the top row, it cannot move up."



```
can_move(Tile, EmptySpace) :- adjacent(Tile, EmptySpace).
adjacent((X1, Y), (X2, Y)) :- abs(X1 - X2) = 1.
adjacent((X, Y1), (X, Y2)) :- abs(Y1 - Y2) = 1.
```

- **Adaptability**:
  - ◦ The agent could apply its reasoning to new tasks or changes in the puzzle. For example, if the rules changed to allow diagonal moves, the agent could incorporate this new rule into its reasoning process.

Advancements in Representation

- **Variable Assignments**:
  - ◦ Representing states as assignments of values to variables is more efficient and allows domain-independent algorithms.
- **Logic as a General Representation**:
  - ◦ Developing logic supports knowledge-based agents.
  - ◦ These agents can combine and recombine information flexibly, handle new tasks, learn new knowledge quickly, and adapt to environmental changes.

KNOWLEDGE-BASED AGENTS

Knowledge Base:

- A knowledge base is a collection of sentences.
- Sentences represent assertions about the world.
- These sentences are expressed in a knowledge representation language.

Example:

- In the context of a smart home, a sentence could be: "The temperature in the living room is 22°C."

- **Axiom**:
  - ◦ An axiom is a sentence taken as given, without derivation from other sentences.
- **Example**:
  - ◦ "All humans are mortal." This is an accepted truth in the system.

Operations: TELL and ASK

- **TELL and ASK**:
  - ◦ TELL: Adds new sentences to the knowledge base.
  - ◦ ASK: Queries the knowledge base to retrieve information.
- **Example**:
  - ◦ TELL: Adding "The kitchen light is on" to the KB.
  - ◦ ASK: Querying "Is the kitchen light on?" should return true based on the KB.
- **Inference**:
  - ◦ The process of deriving new sentences from existing ones.
  - ◦ Ensures that the answers from ASK operations follow logically from the sentences previously TELLed.
- **Example**:
  - ◦ If the KB contains "All birds can fly" and "Tweety is a bird," inference allows the agent to deduce "Tweety can fly."

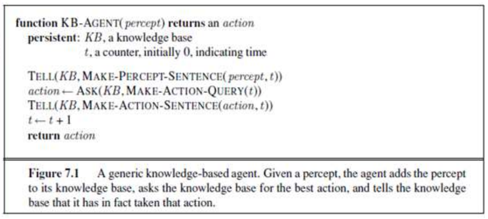
Knowledge-Based Agent Program

Agent Program Outline:

- Percept: Input the agent receives from the environment.
- Action: Output or action the agent performs.
- KB Maintenance: KB contains initial background knowledge and updates with new percepts and actions.

Example:

- A robot vacuum perceives dirt in the living room (percept).
- It queries the KB to decide to vacuum the living room (action).
- After performing the action, it updates the KB with "The living room is clean."

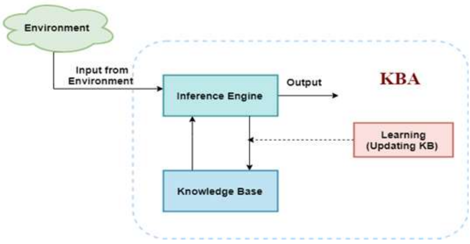


```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  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
```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

- **Functions**:
  - **MAKE-PERCEPT-SENTENCE**: Constructs a sentence from a percept.
  - **MAKE-ACTION-QUERY**: Constructs a sentence to ask what action to take.
  - **MAKE-ACTION-SENTENCE**: Constructs a sentence asserting the action taken.
- **Example**:
  - **MAKE-PERCEPT-SENTENCE**: Converts "see dirt in living room" to "Dirt(living_room)".
  - **MAKE-ACTION-QUERY**: Converts "What should I do at time t?" to "Action(t)".
  - **MAKE-ACTION-SENTENCE**: Converts "vacuum living room at time t" to "Action(vacuum, living_room, t)".

Knowledge Based Agents Architecture



Example Scenario

- Consider a smart home heating system:
- **Input from Environment**: The system receives input that the current temperature in the living room is 18°C.
- **Inference Engine Processing**: The inference engine queries the knowledge base, which contains the preferred temperature range (e.g., 20°C to 22°C).
- **Querying the Knowledge Base**: The inference engine retrieves the preferred temperature range and determines that the current temperature is too low.
- **Output**: The system decides to turn on the heater to raise the temperature.
- **Learning and Updating KB**: Over time, the system learns that the living room cools down faster in the evening and updates the KB with this information to preemptively adjust heating schedules.

Declarative vs. Procedural Approach

- **Declarative Approach**:
  - Sentences are added to the KB to inform the agent about its environment.
  - The agent uses these sentences to reason and act.
- **Example**:
  - Declaratively telling the agent: "The door is locked."
  - The agent knows it needs to unlock the door before exiting.
- **Procedural Approach**:
  - Desired behaviors are encoded directly as program code.
- **Example**:
  - Writing a function directly in code:
  - def exit_room():
    - if door_locked: unlock_door()
    - else: open_door()

Combined Approach

- knowledge-based agent can be built simply by TELLing it what it needs to know. Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment. This is called the declarative approach to system building. In contrast, the procedural approach encodes desired behaviors directly as program code.
- Modern agents use both approaches for efficiency and flexibility.
- Declarative knowledge can be compiled into procedural code for performance.

Example:

- The agent learns from experiences that certain actions always follow specific percepts and compiles this into efficient code.

Learning:

- Agents can have mechanisms to learn from their environment and update their KB autonomously.
- This allows agents to become fully autonomous.

Example:

- A thermostat learns the preferred temperature settings over time based on user adjustments and automatically adjusts the temperature without manual input.

- knowledge-based agents rely on a structured knowledge base and inference mechanisms to reason and act intelligently. They use a combination of declarative and procedural knowledge to achieve their goals efficiently and can learn and adapt autonomously over time.

Logic: Knowledge Bases and Sentences

- **Knowledge Bases (KB):**
  - A knowledge base is a collection of statements (sentences) that an agent knows to be true.
  - These sentences are expressed in a formal representation language.
  - The KB is used by the agent to reason about the world and make decisions.
- **Sentences:** Sentences are the basic units of information in a knowledge base. They are constructed according to the rules of a formal language and represent facts or assertions about the world.

Syntax

- Syntax refers to the rules that define which combinations of symbols are considered valid sentences in the representation language.
- **Example in Arithmetic:**
  - Valid (well-formed) sentence: "x + y = 4"
  - Invalid (not well-formed) sentence: "x4y+="
- In arithmetic, the syntax includes rules about how numbers, variables, and operators can be combined to form valid mathematical statements.

Semantics

- Semantics concerns the meaning of sentences. It defines what it means for a sentence to be true or false in a given context or possible world.
- **Example in Arithmetic:**
  - The sentence "x + y = 4" is true in a world where x is 2 and y is 2.
  - The sentence "x + y = 4" is false in a world where x is 1 and y is 1.

Models and Possible Worlds

- **Possible World:** A possible world is a specific assignment of values to all relevant variables. In logical reasoning, we consider all possible worlds to evaluate the truth of sentences.
- **Model:** A model is a mathematical abstraction that represents a possible world. It specifies the truth or falsehood of each sentence in that world.
- **Example:**
  - Possible world where x = 2 and y = 2 is a model that satisfies the sentence "x + y = 4". X men and Y women sitting at a table playing bridge.
  - Possible world where x = 1 and y = 1 is a model that does not satisfy the sentence "x + y = 4".

Logical entailment

- logical entailment between sentences—the idea that a sentence follows logically from another sentence.
- In mathematical notation, we write

$$\alpha \models \beta$$

- to mean that the sentence α entails the sentence β. The formal definition of entailment is this:
- $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true.
- Using the notation just introduced, we can write

$$\alpha \models \beta \text{ if and only if } M(\alpha) \subseteq M(\beta)\ .$$

Logical Inference

- Logical inference is the process of deriving new sentences from the KB using rules of logic.
- **Inference Algorithm:**
  - Derives conclusions based on entailment.
  - **Model Checking:** Enumerates all possible models to check if α is true in all models where KB is true ($M(KB) \subseteq M(\alpha)$).
- **Soundness:**
  - An inference algorithm is sound if it derives only sentences that are entailed (truth-preserving).
  - **Example:** Model checking is sound because it only confirms conclusions true in all relevant models.

Completeness:

- An inference algorithm is complete if it can derive any sentence that is entailed.
- **Example:** For finite sets, systematic examination ensures completeness, but infinite sets require more sophisticated methods.

Real-World Correspondence

- **Grounding:**
  - Grounding is the connection between logical reasoning processes and the real environment.
  - **Sensors:** Provide percepts that the agent converts into sentences in the KB.
    - Example: A smell sensor detects a wumpus, and the agent adds a sentence about the smell to the KB.
- **Learning:**
  - General rules in the KB are derived from experience and learning processes.
  - Learning can be fallible but aims to create reliable rules about the environment.
    - Example: Learning that wumpuses cause smells, though there might be exceptions (e.g., leap year baths).

Wumpus world

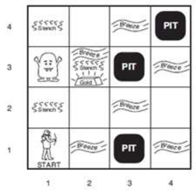


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

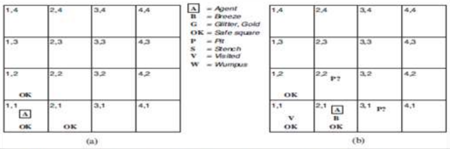


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None, None, None, None, None*]. (b) After one move, with percept [*None, Breeze, None, None, None*].

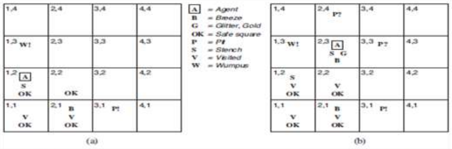


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench, None, None, None, None*]. (b) After the fifth move, with percept [*Stench, Breeze, Glitter, None, None*].

Logic

Logic is the basis of all mathematical reasoning, and of all automated reasoning. The rules of logic specify the meaning of mathematical statements. These rules help us understand and reason with statements such as –

$\exists x$ such that $x \neq a^2 + b^2$, where $x, a, b \in Z$

Which in Simple English means "There exists an integer that is not the sum of two squares". **Importance of Mathematical Logic** The rules of logic give precise meaning to mathematical statements. These rules are used to distinguish between valid and invalid mathematical arguments. Apart from its importance in understanding mathematical reasoning, logic has numerous applications in Computer Science, varying from design of digital circuits, to the construction of computer programs and verification of correctness of programs.

Propositional Logic

A proposition is the basic building block of logic. It is defined as a declarative sentence that is either True or False, but not both. The **Truth Value** of a proposition is True(denoted as T) if it is a true statement, and False(denoted as F) if it is a false statement. For Example,

1. The sun rises in the East and sets in the West.
2. 1 + 1 = 2
3. 'b' is a vowel.

Basic Terminology

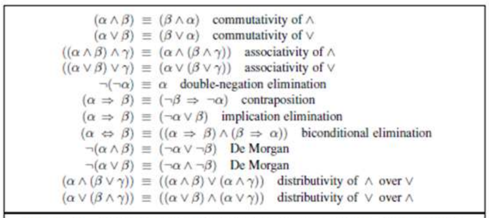
- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- 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**.

Propositional logic

- **Logical constants**: true, false
- **Propositional symbols**: P, Q, S, ... (**atomic sentences**)
- Wrapping **parentheses**: (…)
- Sentences are combined by **connectives**:

∧ ...and	[conjunction]
∨ ...or	[disjunction]
⇒...implies	[implication / conditional]
⇔..is equivalent	[biconditional]
¬ ...not	[negation]

- **Literal**: atomic sentence or negated atomic sentence



$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge$$
$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee$$
$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge$$
$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee$$
$$\neg(\neg\alpha) \equiv \alpha \quad \text{double-negation elimination}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{implication elimination}$$
$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{De Morgan}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{De Morgan}$$
$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee$$
$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge$$

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

Examples of Propositional Logic sentences

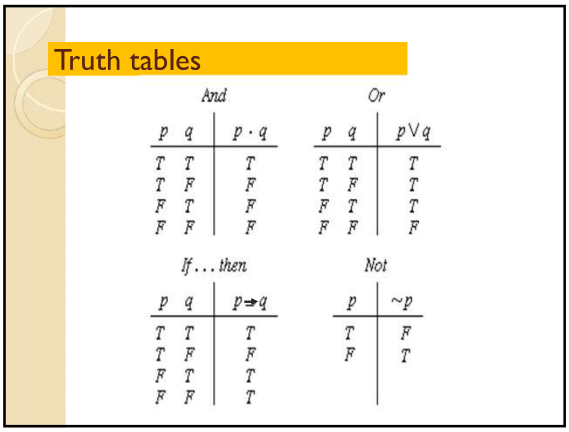
- P means “It is hot.”
- Q means “It is humid.”
- R means “It is raining.”
- (P ∧ Q) → R
  “If it is hot and humid, then it is raining”
- Q → P
  “If it is humid, then it is hot”

Continued…

- A simple language useful for showing key ideas and definitio
- User defines a set of propositional symbols, like P and Q.
- User defines the **semantics** of each propositional symbol:
  - P means “It is hot”
  - Q means “It is humid”
  - R means “It is raining”
- A sentence (well formed formula) is defined as follows:
  - A symbol is a sentence
  - If S is a sentence, then ¬S is a sentence
  - If S is a sentence, then (S) is a sentence
  - If S and T are sentences, then (S ∨ T), (S ∧ T), and (S → T are sentences

Continued…

- A valid sentence or tautology is a sentence that is True under all interpretations, no matter what the world is actually like or how the semantics are defined. Example: “It’s raining or it’s not raining.”
- An inconsistent sentence or contradiction is a sentence that is False under all interpretations. The world is never like what it describes, as in “It’s raining and it’s not raining.”
- P entails Q, written P ⊨ Q, means that whenever P is True, so is Q. In other words, all models of P are also models of Q.



Truth tables

And

p	q	$p \cdot q$
T	T	T
T	F	F
F	T	F
F	F	F

Or

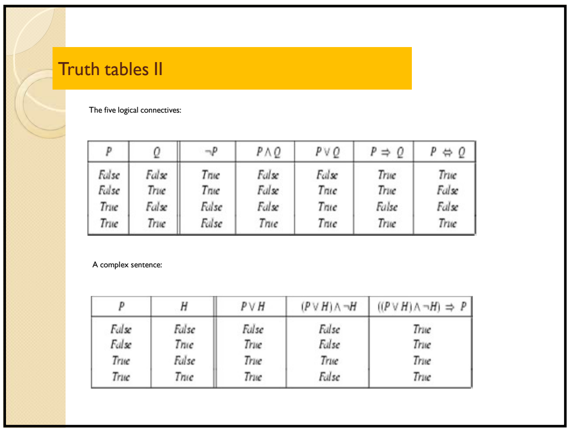
p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

If . . . then

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Not

p	$\sim p$
T	F
F	T



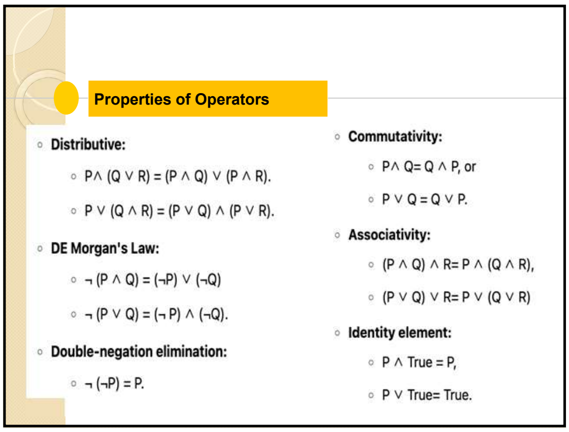
Truth tables II

The five logical connectives:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
False	*False*	*True*	*False*	*False*	*True*	*True*
False	*True*	*True*	*False*	*True*	*True*	*False*
True	*False*	*False*	*False*	*True*	*False*	*False*
True	*True*	*False*	*True*	*True*	*True*	*True*

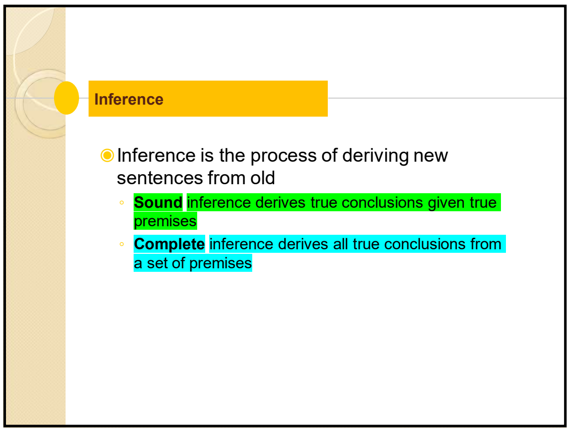
A complex sentence:

P	H	$P \vee H$	$(P \vee H) \wedge \neg H$	$((P \vee H) \wedge \neg H) \Rightarrow P$
False	*False*	*False*	*False*	*True*
False	*True*	*True*	*False*	*True*
True	*False*	*True*	*True*	*True*
True	*True*	*True*	*False*	*True*



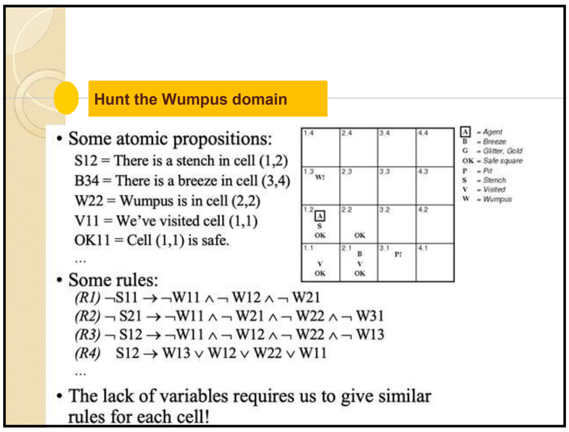
Properties of Operators

- **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.
- **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.



Inference

- Inference is the process of deriving new sentences from old
  - **Sound** inference derives true conclusions given true premises
  - **Complete** inference derives all true conclusions from a set of premises



Hunt the Wumpus domain

- Some atomic propositions:

  S12 = There is a stench in cell (1,2)
  B34 = There is a breeze in cell (3,4)
  W22 = Wumpus is in cell (2,2)
  V11 = We've visited cell (1,1)
  OK11 = Cell (1,1) is safe.
  …

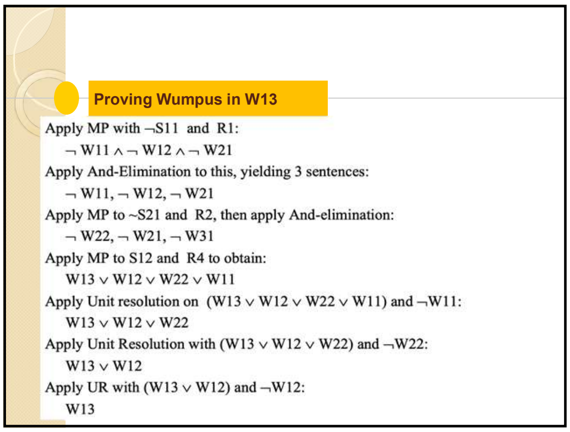
1,4	2,4	3,4	4,4
1,3 W!	2,3	3,3	4,3
1,2 A S OK	2,2 OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
P = Pit
S = Stench
V = Visited
W = Wumpus

- Some rules:

  (R1) ¬S11 → ¬W11 ∧ ¬ W12 ∧ ¬ W21
  (R2) ¬ S21 → ¬W11 ∧ ¬ W21 ∧ ¬ W22 ∧ ¬ W31
  (R3) ¬ S12 → ¬W11 ∧ ¬ W12 ∧ ¬ W22 ∧ ¬ W13
  (R4) S12 → W13 ∨ W12 ∨ W22 ∨ W11
  …

- The lack of variables requires us to give similar rules for each cell!

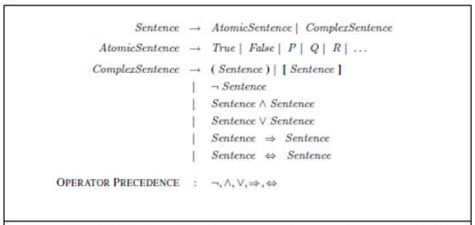


Proving Wumpus in W13

Apply MP with ¬S11 and R1:
¬ W11 ∧ ¬ W12 ∧ ¬ W21
Apply And-Elimination to this, yielding 3 sentences:
¬ W11, ¬ W12, ¬ W21
Apply MP to ~S21 and R2, then apply And-elimination:
¬ W22, ¬ W21, ¬ W31
Apply MP to S12 and R4 to obtain:
W13 ∨ W12 ∨ W22 ∨ W11
Apply Unit resolution on (W13 ∨ W12 ∨ W22 ∨ W11) and ¬W11:
W13 ∨ W12 ∨ W22
Apply Unit Resolution with (W13 ∨ W12 ∨ W22) and ¬W22:
W13 ∨ W12
Apply UR with (W13 ∨ W12) and ¬W12:
W13

1. Syntax of Propositional Logic:

- **Atomic Sentences:** The simplest form of sentences in propositional logic, consisting of *proposition symbols* like P, Q, R, or specific symbols like W1,3. These symbols represent propositions that can be either true or false.
  - Example: W1,3 might represent "The Wumpus is in [1,3]."
- **Complex Sentences:** Formed by combining atomic sentences using logical connectives such as AND (∧), OR (∨), NOT (¬), IMPLIES (⇒), and IF AND ONLY IF (⇔).



```
           Sentence  →  AtomicSentence | ComplexSentence
     AtomicSentence  →  True | False | P | Q | R | ...
    ComplexSentence  →  ( Sentence ) | [ Sentence ]
                     |  ¬ Sentence
                     |  Sentence ∧ Sentence
                     |  Sentence ∨ Sentence
                     |  Sentence  ⇒  Sentence
                     |  Sentence  ⇔  Sentence

OPERATOR PRECEDENCE  :  ¬, ∧, ∨, ⇒, ⇔
```

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

- **Negation (¬):** Inverts the truth value of an atomic sentence.
  - Example: ¬W1,3 means "The Wumpus is not in [1,3]."
- **Conjunction (∧):** True if both connected sentences are true.
  - Example: W1,3 ∧ P3,1 means "The Wumpus is in [1,3] and there's a pit in [3,1]."
- **Disjunction (∨):** True if at least one of the connected sentences is true.
  - Example: (W1,3 ∧ P3,1) ∨ W2,2 means "Either the Wumpus is in [1,3] and there's a pit in [3,1], or there's a Wumpus in [2,2]."
- **Implication (⇒):** True unless the first sentence (antecedent) is true and the second (consequent) is false.
  - Example: (W1,3 ∧ P3,1) ⇒ ¬W2,2 means "If the Wumpus is in [1,3] and there's a pit in [3,1], then the Wumpus is not in [2,2]."
- **Biconditional (⇔):** True if both sentences have the same truth value.
  - Example: W1,3 ⇔ ¬W2,2 means "The Wumpus is in [1,3] if and only if the Wumpus is not in [2,2]."

2. Semantics of Propositional Logic

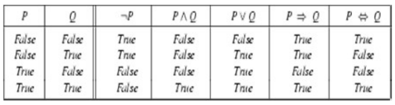
- **Model:** A model in propositional logic assigns a truth value (true or false) to each proposition symbol.
  - Example: A model might specify P1,2 = false, P2,2 = false, P3,1 = true.
- **Truth of Atomic Sentences:** Determined directly by the model. For example, if the model says P1,2 = false, then P1,2 is false.
- **Truth of Complex Sentences:** Determined recursively using the truth values of atomic sentences and the rules for the logical connectives.
  - **Negation (¬P):** True if P is false in the model.
  - **Conjunction (P ∧ Q):** True if both P and Q are true in the model.
  - **Disjunction (P ∨ Q):** True if at least one of P or Q is true in the model.
  - **Implication (P ⇒ Q):** True unless P is true and Q is false.
  - **Biconditional (P ⇔ Q):** True if P and Q are both true or both false.

3. Examples of Truth Evaluation

- Given a model m1 = {P1,2 = false, P2,2 = false, P3,1 = true}, evaluate the complex sentence ¬P1,2 ∧ (P2,2 ∨ P3,1):
  - ¬P1,2 is true because P1,2 is false in m1.
  - P2,2 ∨ P3,1 is true because P3,1 is true in m1.
  - Therefore, ¬P1,2 ∧ (P2,2 ∨ P3,1) is true.

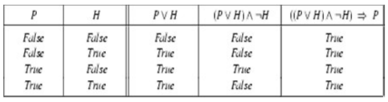
Truth tables II

The five logical connectives:



P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
False	*False*	*True*	*False*	*False*	*True*	*True*
False	*True*	*True*	*False*	*True*	*True*	*False*
True	*False*	*False*	*False*	*True*	*False*	*False*
True	*True*	*False*	*True*	*True*	*True*	*True*

A complex sentence:



P	H	$P \vee H$	$(P \vee H) \wedge \neg H$	$((P \vee H) \wedge \neg H) \Rightarrow P$
False	*False*	*False*	*False*	*True*
False	*True*	*True*	*False*	*True*
True	*False*	*True*	*True*	*True*
True	*True*	*True*	*False*	*True*

A Simple Knowledge Base for the Wumpus World:

- The knowledge base in the Wumpus World consists of sentences that describe both immutable aspects (facts that do not change) and mutable aspects (facts that can change) of the environment.
- **Symbols:**
- **Px,y**: True if there is a pit in square [x,y].
- **Wx,y**: True if there is a Wumpus in square [x,y], dead or alive.
- **Bx,y**: True if the agent perceives a breeze in square [x,y.
- **Sx,y**: True if the agent perceives a stench in square [x,y]
- These symbols are used to create logical sentences that describe the state of the Wumpus World.

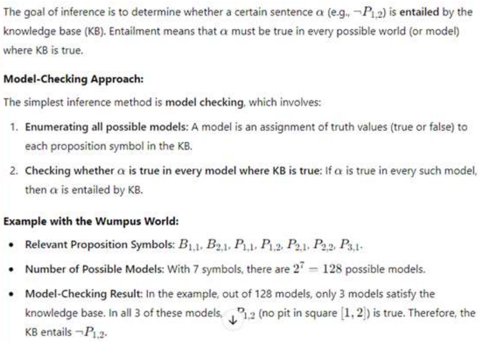
Example Sentences:

- **R1: ¬P1,1** – There is no pit in square [1,1]
- **R2: B1,1 ⇔ (P1,2 ∨ P2,1)** – Square [1,1] is breezy if and only if there is a pit in one of its neighboring squares [1,2]or [2,1]
- **R3: B2,1 ⇔ (P1,1 ∨ P2,2 ∨ P3,1)** – Square [2,1] is breezy if and only if there is a pit in one of its neighboring squares [1,1] [2,2]or [3,1]

Percepts in a Specific World:

- **R4: ¬B1,1** – There is no breeze in square [1,1].
- **R5: B2,1** – There is a breeze in square [2,1].
- These sentences together describe the agent's knowledge of the Wumpus World, combining general rules (like the relationship between breezes and pits) with specific percepts from the current environment.

Inference in Propositional Logic



The goal of inference is to determine whether a certain sentence α (e.g., $\neg P_{1,2}$) is **entailed** by the knowledge base (KB). Entailment means that α must be true in every possible world (or model) where KB is true.

Model-Checking Approach:

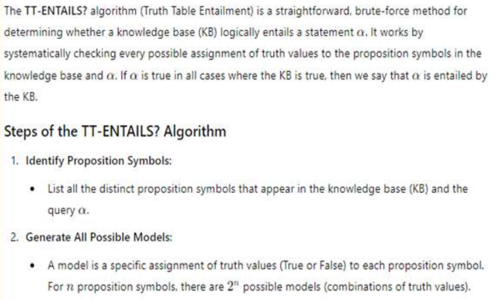
The simplest inference method is **model checking**, which involves:

1. **Enumerating all possible models:** A model is an assignment of truth values (true or false) to each proposition symbol in the KB.
2. **Checking whether α is true in every model where KB is true:** If α is true in every such model, then α is entailed by KB.

Example with the Wumpus World:

- **Relevant Proposition Symbols:** $B_{1,1}, B_{2,1}, P_{1,1}, P_{1,2}, P_{2,1}, P_{2,2}, P_{3,1}$.
- **Number of Possible Models:** With 7 symbols, there are $2^7 = 128$ possible models.
- **Model-Checking Result:** In the example, out of 128 models, only 3 models satisfy the knowledge base. In all 3 of these models, ↓$P_{1,2}$ (no pit in square $[1, 2]$) is true. Therefore, the KB entails $\neg P_{1,2}$.

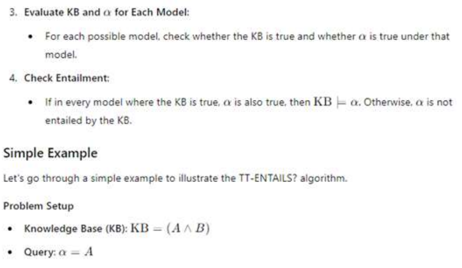
TT-entails algorithm



The **TT-ENTAILS?** algorithm (Truth Table Entailment) is a straightforward, brute-force method for determining whether a knowledge base (KB) logically entails a statement α. It works by systematically checking every possible assignment of truth values to the proposition symbols in the knowledge base and α. If α is true in all cases where the KB is true, then we say that α is entailed by the KB.

Steps of the TT-ENTAILS? Algorithm

1. **Identify Proposition Symbols:**
   - List all the distinct proposition symbols that appear in the knowledge base (KB) and the query α.
2. **Generate All Possible Models:**
   - A model is a specific assignment of truth values (True or False) to each proposition symbol. For n proposition symbols, there are 2^n possible models (combinations of truth values).



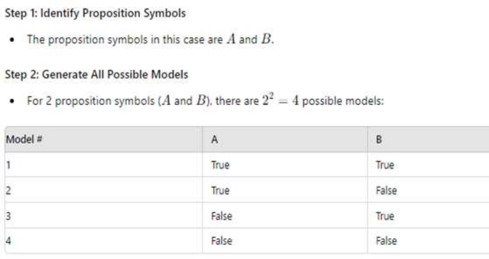
3. **Evaluate KB and α for Each Model:**
   - For each possible model, check whether the KB is true and whether α is true under that model.
4. **Check Entailment:**
   - If in every model where the KB is true, α is also true, then $\mathrm{KB} \models \alpha$. Otherwise, α is not entailed by the KB.

Simple Example

Let's go through a simple example to illustrate the TT-ENTAILS? algorithm.

Problem Setup

- **Knowledge Base (KB):** $\mathrm{KB} = (A \wedge B)$
- **Query:** $\alpha = A$



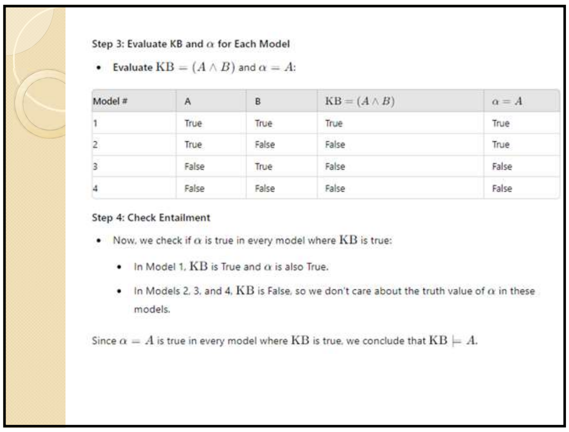
Step 1: Identify Proposition Symbols

- The proposition symbols in this case are A and B.

Step 2: Generate All Possible Models

- For 2 proposition symbols (A and B), there are $2^2 = 4$ possible models:

Model #	A	B
1	True	True
2	True	False
3	False	True
4	False	False



Step 3: Evaluate KB and α for Each Model

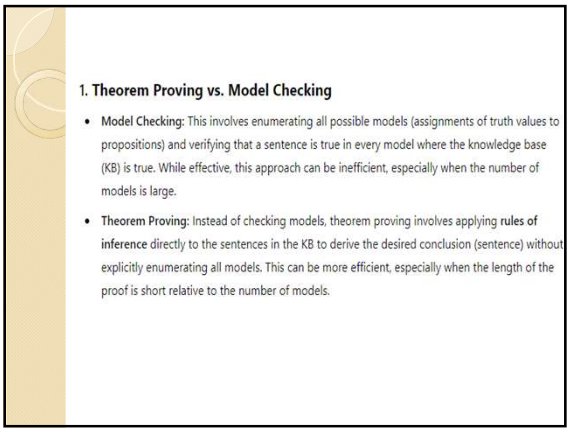
- **Evaluate $\mathrm{KB} = (A \wedge B)$ and $\alpha = A$:**

Model #	A	B	$\mathrm{KB} = (A \wedge B)$	$\alpha = A$
1	True	True	True	True
2	True	False	False	True
3	False	True	False	False
4	False	False	False	False

Step 4: Check Entailment

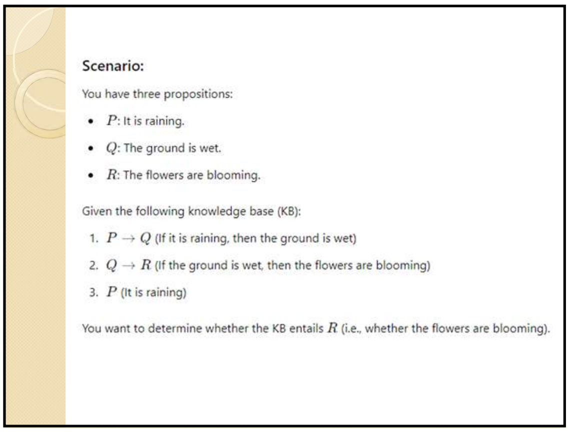
- Now, we check if α is true in every model where KB is true:
  - In Model 1, KB is True and α is also True.
  - In Models 2, 3, and 4, KB is False, so we don't care about the truth value of α in these models.

Since $\alpha = A$ is true in every model where KB is true, we conclude that $\mathrm{KB} \models A$.



1. Theorem Proving vs. Model Checking

- **Model Checking:** This involves enumerating all possible models (assignments of truth values to propositions) and verifying that a sentence is true in every model where the knowledge base (KB) is true. While effective, this approach can be inefficient, especially when the number of models is large.
- **Theorem Proving:** Instead of checking models, theorem proving involves applying **rules of inference** directly to the sentences in the KB to derive the desired conclusion (sentence) without explicitly enumerating all models. This can be more efficient, especially when the length of the proof is short relative to the number of models.



Scenario:

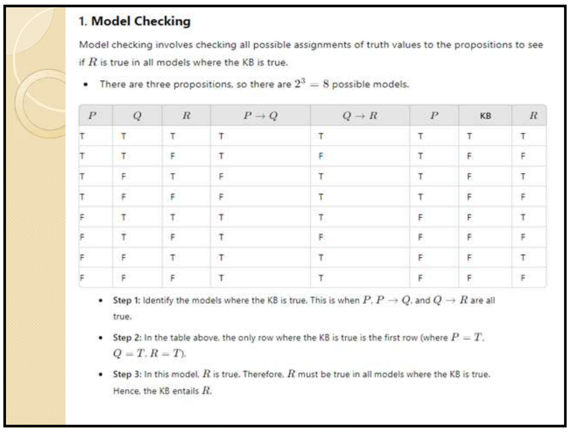
You have three propositions:

- P: It is raining.
- Q: The ground is wet.
- R: The flowers are blooming.

Given the following knowledge base (KB):

1. $P \rightarrow Q$ (If it is raining, then the ground is wet)
2. $Q \rightarrow R$ (If the ground is wet, then the flowers are blooming)
3. P (It is raining)

You want to determine whether the KB entails R (i.e., whether the flowers are blooming).



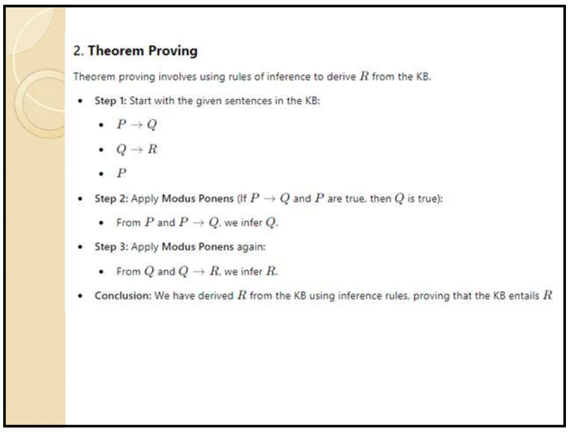
1. Model Checking

Model checking involves checking all possible assignments of truth values to the propositions to see if R is true in all models where the KB is true.

- There are three propositions, so there are $2^3 = 8$ possible models.

P	Q	R	$P \rightarrow Q$	$Q \rightarrow R$	P	KB	R
T	T	T	T	T	T	T	T
T	T	F	T	F	T	F	F
T	F	T	F	T	T	F	T
T	F	F	F	T	T	F	F
F	T	T	T	T	F	F	T
F	T	F	T	F	F	F	F
F	F	T	T	T	F	F	T
F	F	F	T	T	F	F	F

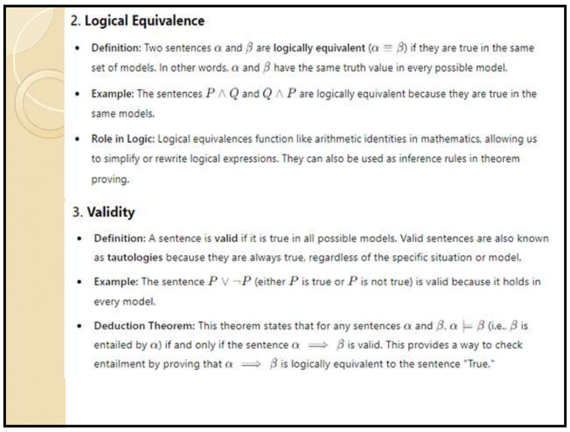
- **Step 1:** Identify the models where the KB is true. This is when P, $P \rightarrow Q$, and $Q \rightarrow R$ are all true.
- **Step 2:** In the table above, the only row where the KB is true is the first row (where $P = T$, $Q = T$, $R = T$).
- **Step 3:** In this model, R is true. Therefore, R must be true in all models where the KB is true. Hence, the KB entails R.



2. Theorem Proving

Theorem proving involves using rules of inference to derive R from the KB.

- **Step 1:** Start with the given sentences in the KB:
  - $P \rightarrow Q$
  - $Q \rightarrow R$
  - P
- **Step 2:** Apply **Modus Ponens** (If $P \rightarrow Q$ and P are true, then Q is true):
  - From P and $P \rightarrow Q$, we infer Q.
- **Step 3:** Apply **Modus Ponens** again:
  - From Q and $Q \rightarrow R$, we infer R.
- **Conclusion:** We have derived R from the KB using inference rules, proving that the KB entails R

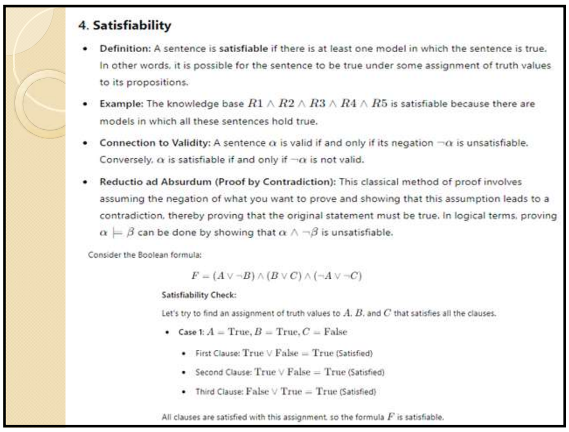


2. Logical Equivalence

- **Definition:** Two sentences α and β are **logically equivalent** ($\alpha \equiv \beta$) if they are true in the same set of models. In other words, α and β have the same truth value in every possible model.
- **Example:** The sentences $P \wedge Q$ and $Q \wedge P$ are logically equivalent because they are true in the same models.
- **Role in Logic:** Logical equivalences function like arithmetic identities in mathematics, allowing us to simplify or rewrite logical expressions. They can also be used as inference rules in theorem proving.

3. Validity

- **Definition:** A sentence is **valid** if it is true in all possible models. Valid sentences are also known as **tautologies** because they are always true, regardless of the specific situation or model.
- **Example:** The sentence $P \vee \neg P$ (either P is true or P is not true) is valid because it holds in every model.
- **Deduction Theorem:** This theorem states that for any sentences α and β, $\alpha \models \beta$ (i.e., β is entailed by α) if and only if the sentence $\alpha \implies \beta$ is valid. This provides a way to check entailment by proving that $\alpha \implies \beta$ is logically equivalent to the sentence "True."



4. Satisfiability

- **Definition:** A sentence is **satisfiable** if there is at least one model in which the sentence is true. In other words, it is possible for the sentence to be true under some assignment of truth values to its propositions.
- **Example:** The knowledge base $R1 \wedge R2 \wedge R3 \wedge R4 \wedge R5$ is satisfiable because there are models in which all these sentences hold true.
- **Connection to Validity:** A sentence α is valid if and only if its negation $\neg\alpha$ is unsatisfiable. Conversely, α is satisfiable if and only if $\neg\alpha$ is not valid.
- **Reductio ad Absurdum (Proof by Contradiction):** This classical method of proof involves assuming the negation of what you want to prove and showing that this assumption leads to a contradiction, thereby proving that the original statement must be true. In logical terms, proving $\alpha \models \beta$ can be done by showing that $\alpha \wedge \neg\beta$ is unsatisfiable.

Consider the Boolean formula:

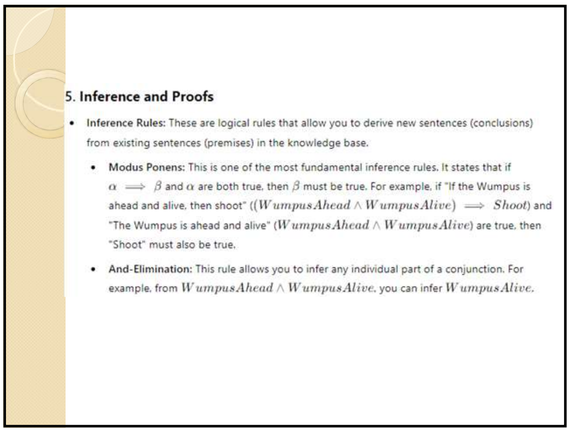
$$F = (A \vee \neg B) \wedge (B \vee C) \wedge (\neg A \vee \neg C)$$

Satisfiability Check:

Let's try to find an assignment of truth values to A, B, and C that satisfies all the clauses.

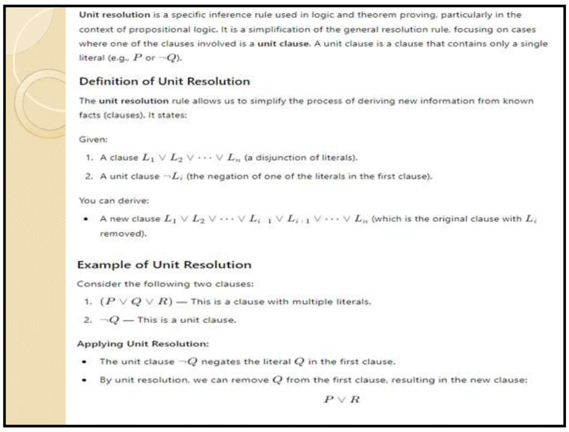
- **Case 1:** $A = \text{True}, B = \text{True}, C = \text{False}$
  - First Clause: $\text{True} \vee \text{False} = \text{True}$ (Satisfied)
  - Second Clause: $\text{True} \vee \text{False} = \text{True}$ (Satisfied)
  - Third Clause: $\text{False} \vee \text{True} = \text{True}$ (Satisfied)

All clauses are satisfied with this assignment, so the formula F is satisfiable.



5. Inference and Proofs

- **Inference Rules:** These are logical rules that allow you to derive new sentences (conclusions) from existing sentences (premises) in the knowledge base.
  - **Modus Ponens:** This is one of the most fundamental inference rules. It states that if $\alpha \implies \beta$ and α are both true, then β must be true. For example, if "If the Wumpus is ahead and alive, then shoot" ($(WumpusAhead \wedge WumpusAlive) \implies Shoot$) and "The Wumpus is ahead and alive" ($WumpusAhead \wedge WumpusAlive$) are true, then "Shoot" must also be true.
  - **And-Elimination:** This rule allows you to infer any individual part of a conjunction. For example, from $WumpusAhead \wedge WumpusAlive$, you can infer $WumpusAlive$.



Unit resolution is a specific inference rule used in logic and theorem proving, particularly in the context of propositional logic. It is a simplification of the general resolution rule, focusing on cases where one of the clauses involved is a **unit clause.** A unit clause is a clause that contains only a single literal (e.g., P or $\neg Q$).

Definition of Unit Resolution

The **unit resolution** rule allows us to simplify the process of deriving new information from known facts (clauses). It states:

Given:

1. A clause $L_1 \vee L_2 \vee \cdots \vee L_n$ (a disjunction of literals).
2. A unit clause $\neg L_i$ (the negation of one of the literals in the first clause).

You can derive:

- A new clause $L_1 \vee L_2 \vee \cdots \vee L_{i-1} \vee L_{i+1} \vee \cdots \vee L_n$ (which is the original clause with L_i removed).

Example of Unit Resolution

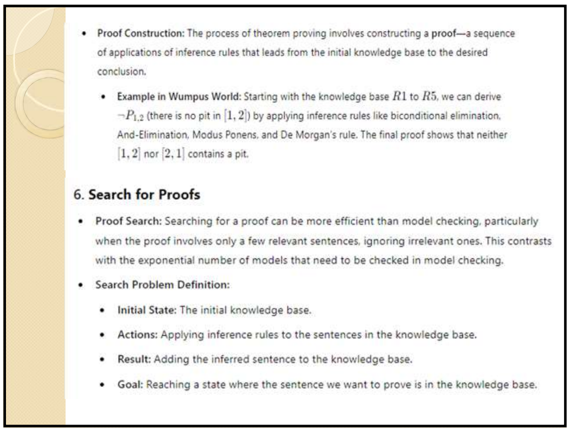
Consider the following two clauses:

1. $(P \vee Q \vee R)$ — This is a clause with multiple literals.
2. $\neg Q$ — This is a unit clause.

Applying Unit Resolution:

- The unit clause $\neg Q$ negates the literal Q in the first clause.
- By unit resolution, we can remove Q from the first clause, resulting in the new clause:

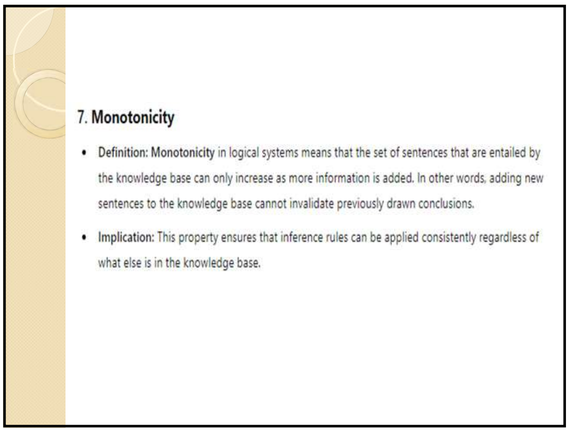
$$P \vee R$$



- **Proof Construction:** The process of theorem proving involves constructing a **proof—**a sequence of applications of inference rules that leads from the initial knowledge base to the desired conclusion.
  - **Example in Wumpus World:** Starting with the knowledge base $R1$ to $R5$, we can derive $\neg P_{1,2}$ (there is no pit in $[1,2]$) by applying inference rules like biconditional elimination, And-Elimination, Modus Ponens, and De Morgan's rule. The final proof shows that neither $[1,2]$ nor $[2,1]$ contains a pit.

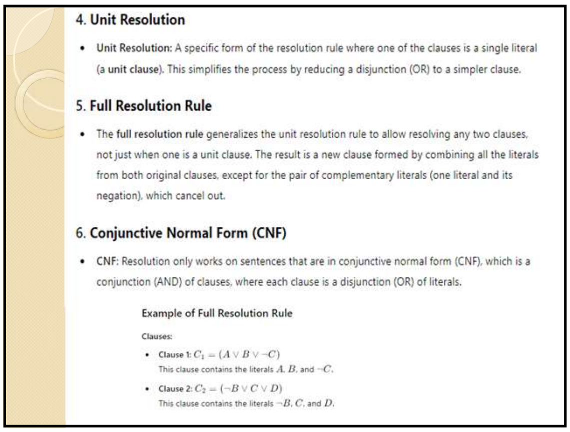
6. Search for Proofs

- **Proof Search:** Searching for a proof can be more efficient than model checking, particularly when the proof involves only a few relevant sentences, ignoring irrelevant ones. This contrasts with the exponential number of models that need to be checked in model checking.
- Search Problem Definition:
  - **Initial State:** The initial knowledge base.
  - **Actions:** Applying inference rules to the sentences in the knowledge base.
  - **Result:** Adding the inferred sentence to the knowledge base.
  - **Goal:** Reaching a state where the sentence we want to prove is in the knowledge base.



7. Monotonicity

- **Definition: Monotonicity** in logical systems means that the set of sentences that are entailed by the knowledge base can only increase as more information is added. In other words, adding new sentences to the knowledge base cannot invalidate previously drawn conclusions.
- **Implication:** This property ensures that inference rules can be applied consistently regardless of what else is in the knowledge base.



4. Unit Resolution

- **Unit Resolution:** A specific form of the resolution rule where one of the clauses is a single literal (a **unit clause**). This simplifies the process by reducing a disjunction (OR) to a simpler clause.

5. Full Resolution Rule

- The **full resolution rule** generalizes the unit resolution rule to allow resolving any two clauses, not just when one is a unit clause. The result is a new clause formed by combining all the literals from both original clauses, except for the pair of complementary literals (one literal and its negation), which cancel out.

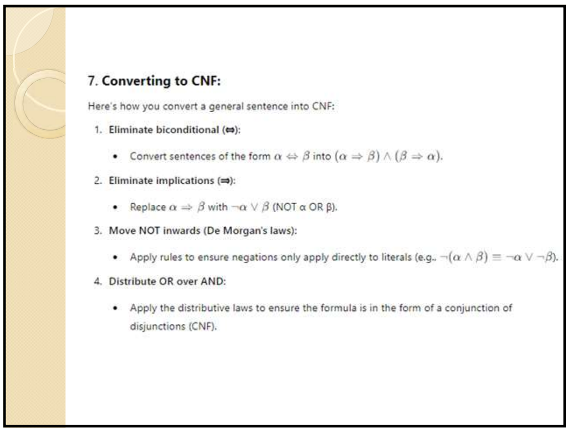
6. Conjunctive Normal Form (CNF)

- CNF: Resolution only works on sentences that are in conjunctive normal form (CNF), which is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals.

Example of Full Resolution Rule

Clauses:

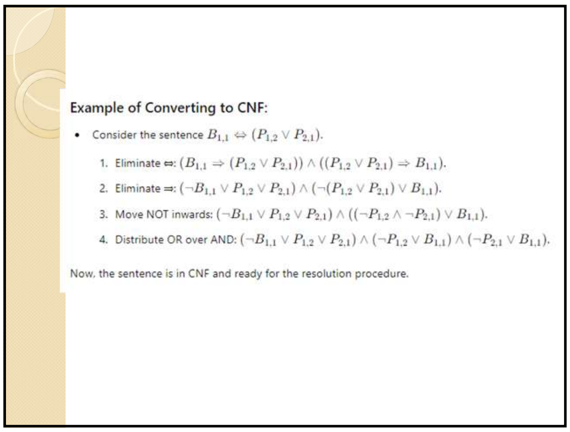
- **Clause 1:** $C_1 = (A \vee B \vee \neg C)$
  This clause contains the literals A, B, and $\neg C$.
- **Clause 2:** $C_2 = (\neg B \vee C \vee D)$
  This clause contains the literals $\neg B$, C, and D.



7. Converting to CNF:

Here's how you convert a general sentence into CNF:

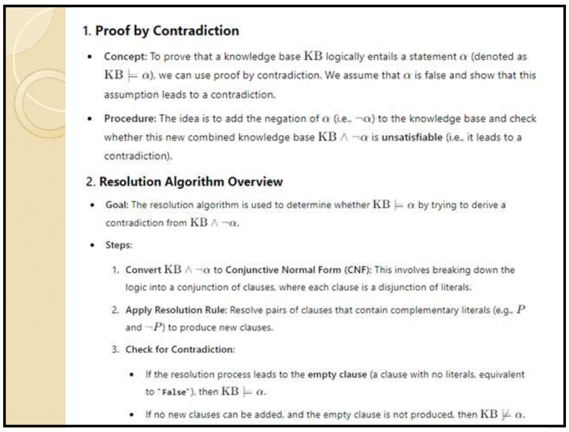
1. **Eliminate biconditional (⇔):**
   - Convert sentences of the form $\alpha \Leftrightarrow \beta$ into $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
2. **Eliminate implications (⇒):**
   - Replace $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$ (NOT α OR β).
3. **Move NOT inwards (De Morgan's laws):**
   - Apply rules to ensure negations only apply directly to literals (e.g., $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$).
4. **Distribute OR over AND:**
   - Apply the distributive laws to ensure the formula is in the form of a conjunction of disjunctions (CNF).



Example of Converting to CNF:

- Consider the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.
  1. Eliminate ⇔: $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$.
  2. Eliminate ⇒: $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$.
  3. Move NOT inwards: $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$.
  4. Distribute OR over AND: $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$.

Now, the sentence is in CNF and ready for the resolution procedure.

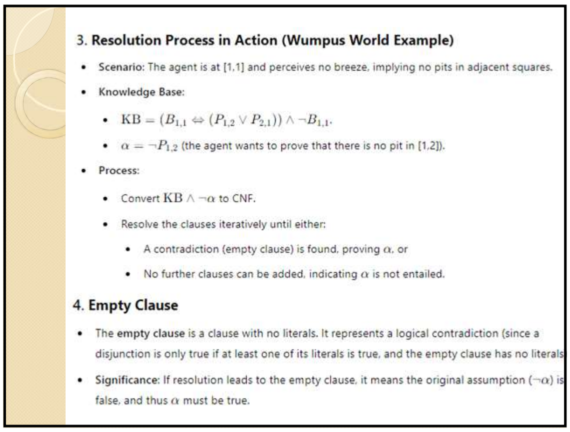


1. Proof by Contradiction

- **Concept:** To prove that a knowledge base KB logically entails a statement α (denoted as $\mathrm{KB} \models \alpha$), we can use proof by contradiction. We assume that α is false and show that this assumption leads to a contradiction.
- **Procedure:** The idea is to add the negation of α (i.e., $\neg\alpha$) to the knowledge base and check whether this new combined knowledge base $\mathrm{KB} \wedge \neg\alpha$ is **unsatisfiable** (i.e., it leads to a contradiction).

2. Resolution Algorithm Overview

- **Goal:** The resolution algorithm is used to determine whether $\mathrm{KB} \models \alpha$ by trying to derive a contradiction from $\mathrm{KB} \wedge \neg\alpha$.
- **Steps:**
  1. **Convert $\mathrm{KB} \wedge \neg\alpha$ to Conjunctive Normal Form (CNF):** This involves breaking down the logic into a conjunction of clauses, where each clause is a disjunction of literals.
  2. **Apply Resolution Rule:** Resolve pairs of clauses that contain complementary literals (e.g., P and $\neg P$) to produce new clauses.
  3. **Check for Contradiction:**
     - If the resolution process leads to the **empty clause** (a clause with no literals, equivalent to `` `False` ``), then $\mathrm{KB} \models \alpha$.
     - If no new clauses can be added, and the empty clause is not produced, then $\mathrm{KB} \not\models \alpha$.

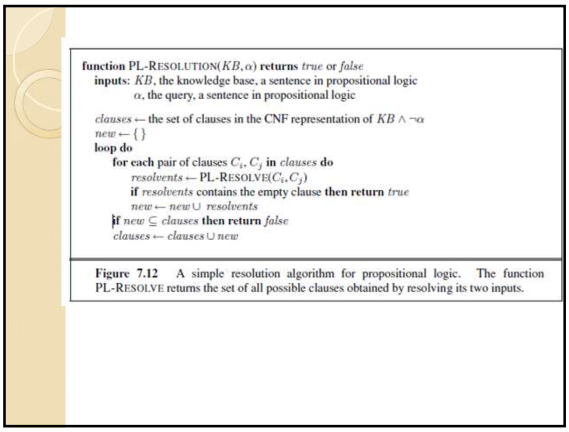


3. Resolution Process in Action (Wumpus World Example)

- **Scenario:** The agent is at [1,1] and perceives no breeze, implying no pits in adjacent squares.
- **Knowledge Base:**
  - $\mathrm{KB} = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$.
  - $\alpha = \neg P_{1,2}$ (the agent wants to prove that there is no pit in [1,2]).
- **Process:**
  - Convert $\mathrm{KB} \wedge \neg\alpha$ to CNF.
  - Resolve the clauses iteratively until either:
    - A contradiction (empty clause) is found, proving α, or
    - No further clauses can be added, indicating α is not entailed.

4. Empty Clause

- The **empty clause** is a clause with no literals. It represents a logical contradiction (since a disjunction is only true if at least one of its literals is true, and the empty clause has no literals
- **Significance:** If resolution leads to the empty clause, it means the original assumption ($\neg\alpha$) is false, and thus α must be true.



```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
     for each pair of clauses C_i, C_j in clauses do
        resolvents ← PL-RESOLVE(C_i, C_j)
        if resolvents contains the empty clause then return true
        new ← new ∪ resolvents
     if new ⊆ clauses then return false
     clauses ← clauses ∪ new
```

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

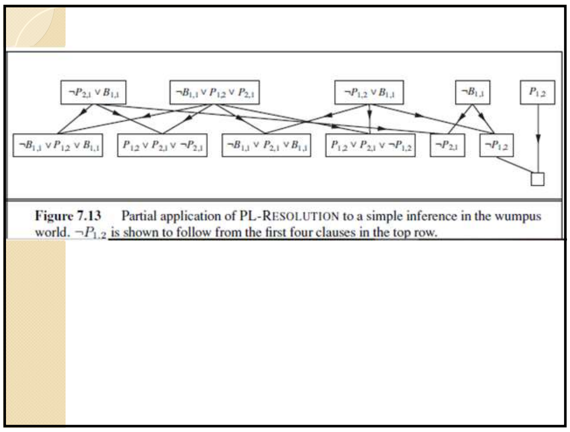
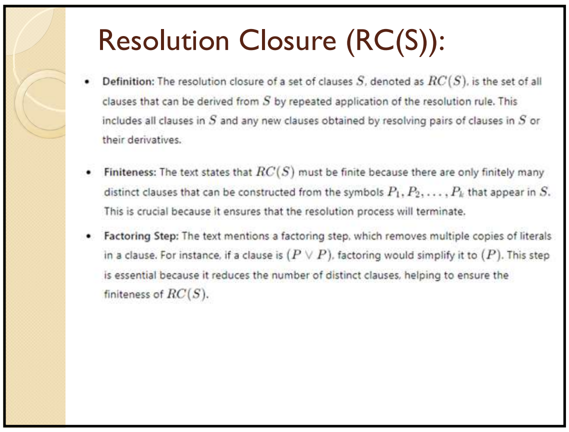
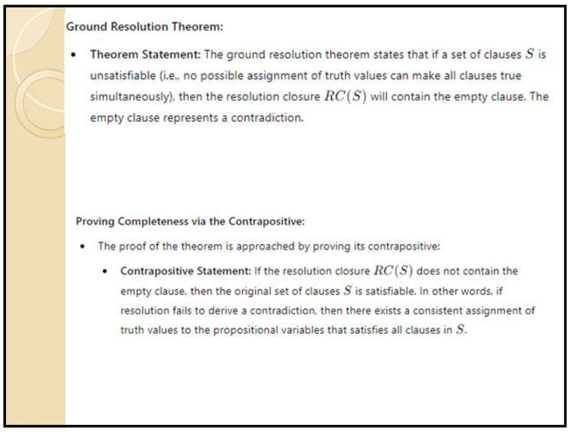

$\neg P_{2,1} \vee B_{1,1}$
$\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$
$\neg P_{1,2} \vee B_{1,1}$
$\neg B_{1,1}$
$P_{1,2}$
$\neg B_{1,1} \vee P_{1,2} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{2,1}$
$\neg B_{1,1} \vee P_{2,1} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{1,2}$
$\neg P_{2,1}$
$\neg P_{1,2}$

Figure 7.13 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.



Resolution Closure (RC(S)):

- **Definition:** The resolution closure of a set of clauses S, denoted as $RC(S)$, is the set of all clauses that can be derived from S by repeated application of the resolution rule. This includes all clauses in S and any new clauses obtained by resolving pairs of clauses in S or their derivatives.
- **Finiteness:** The text states that $RC(S)$ must be finite because there are only finitely many distinct clauses that can be constructed from the symbols $P_1, P_2, \ldots, P_k$ that appear in S. This is crucial because it ensures that the resolution process will terminate.
- **Factoring Step:** The text mentions a factoring step, which removes multiple copies of literals in a clause. For instance, if a clause is $(P \lor P)$, factoring would simplify it to (P). This step is essential because it reduces the number of distinct clauses, helping to ensure the finiteness of $RC(S)$.

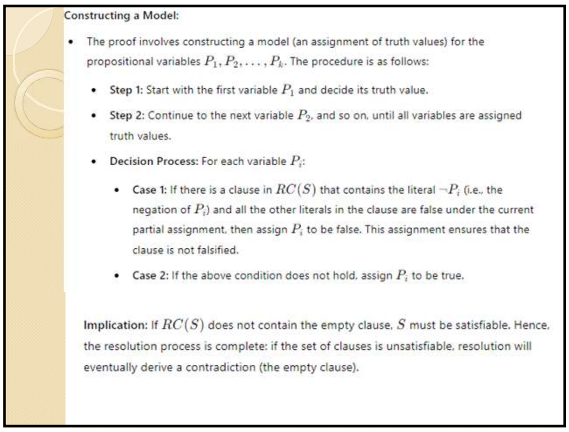


Ground Resolution Theorem:

- **Theorem Statement:** The ground resolution theorem states that if a set of clauses S is unsatisfiable (i.e., no possible assignment of truth values can make all clauses true simultaneously), then the resolution closure $RC(S)$ will contain the empty clause. The empty clause represents a contradiction.

Proving Completeness via the Contrapositive:

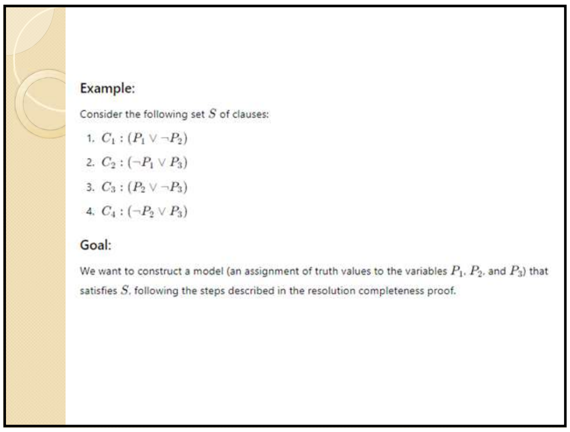
- The proof of the theorem is approached by proving its contrapositive:
  - **Contrapositive Statement:** If the resolution closure $RC(S)$ does not contain the empty clause, then the original set of clauses S is satisfiable. In other words, if resolution fails to derive a contradiction, then there exists a consistent assignment of truth values to the propositional variables that satisfies all clauses in S.



Constructing a Model:

- The proof involves constructing a model (an assignment of truth values) for the propositional variables $P_1, P_2, \ldots, P_k$. The procedure is as follows:
  - **Step 1:** Start with the first variable P_1 and decide its truth value.
  - **Step 2:** Continue to the next variable P_2, and so on, until all variables are assigned truth values.
  - **Decision Process:** For each variable P_i:
    - **Case 1:** If there is a clause in $RC(S)$ that contains the literal $\neg P_i$ (i.e., the negation of P_i) and all the other literals in the clause are false under the current partial assignment, then assign P_i to be false. This assignment ensures that the clause is not falsified.
    - **Case 2:** If the above condition does not hold, assign P_i to be true.

Implication: If $RC(S)$ does not contain the empty clause, S must be satisfiable. Hence, the resolution process is complete: if the set of clauses is unsatisfiable, resolution will eventually derive a contradiction (the empty clause).



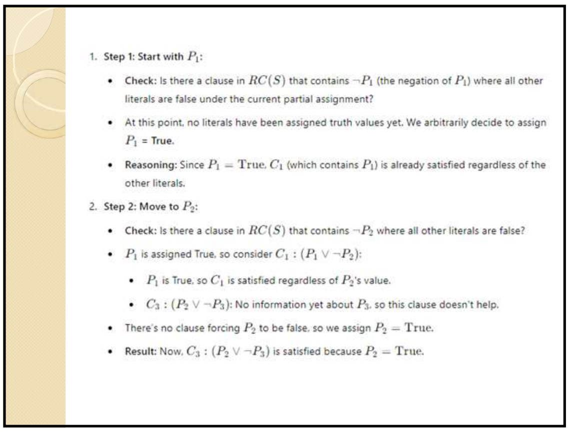
Example:

Consider the following set S of clauses:

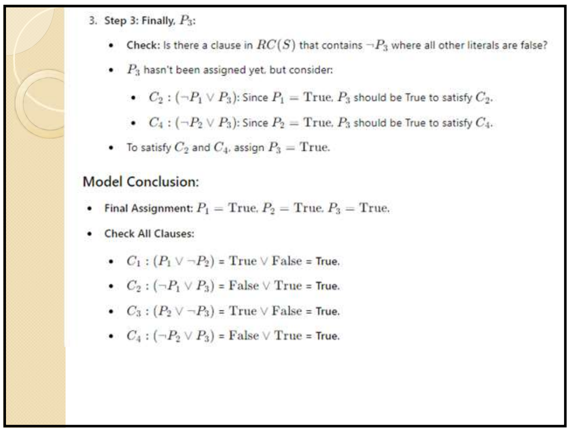
1. $C_1 : (P_1 \lor \neg P_2)$
2. $C_2 : (\neg P_1 \lor P_3)$
3. $C_3 : (P_2 \lor \neg P_3)$
4. $C_4 : (\neg P_2 \lor P_3)$

Goal:

We want to construct a model (an assignment of truth values to the variables P_1, P_2, and P_3) that satisfies S, following the steps described in the resolution completeness proof.



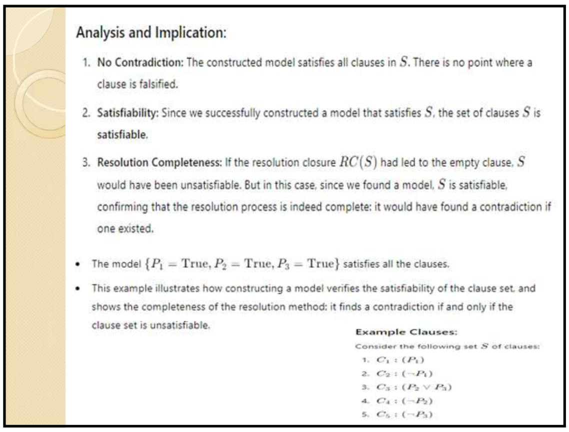
1. **Step 1: Start with P_1:**
   - **Check:** Is there a clause in $RC(S)$ that contains $\neg P_1$ (the negation of P_1) where all other literals are false under the current partial assignment?
   - At this point, no literals have been assigned truth values yet. We arbitrarily decide to assign P_1 = **True.**
   - **Reasoning:** Since $P_1 = \text{True}$, C_1 (which contains P_1) is already satisfied regardless of the other literals.
2. **Step 2: Move to P_2:**
   - **Check:** Is there a clause in $RC(S)$ that contains $\neg P_2$ where all other literals are false?
   - P_1 is assigned True, so consider $C_1 : (P_1 \lor \neg P_2)$:
     - P_1 is True, so C_1 is satisfied regardless of P_2's value.
     - $C_3 : (P_2 \lor \neg P_3)$: No information yet about P_3, so this clause doesn't help.
   - There's no clause forcing P_2 to be false, so we assign $P_2 = \text{True}$.
   - **Result:** Now, $C_3 : (P_2 \lor \neg P_3)$ is satisfied because $P_2 = \text{True}$.



3. **Step 3: Finally, P_3:**
   - **Check:** Is there a clause in $RC(S)$ that contains $\neg P_3$ where all other literals are false?
   - P_3 hasn't been assigned yet, but consider:
     - $C_2 : (\neg P_1 \lor P_3)$: Since $P_1 = \text{True}$, P_3 should be True to satisfy C_2.
     - $C_4 : (\neg P_2 \lor P_3)$: Since $P_2 = \text{True}$, P_3 should be True to satisfy C_4.
   - To satisfy C_2 and C_4, assign $P_3 = \text{True}$.

Model Conclusion:

- **Final Assignment:** $P_1 = \text{True}$, $P_2 = \text{True}$, $P_3 = \text{True}$.
- **Check All Clauses:**
  - $C_1 : (P_1 \lor \neg P_2)$ = $\text{True} \lor \text{False}$ = **True.**
  - $C_2 : (\neg P_1 \lor P_3)$ = $\text{False} \lor \text{True}$ = **True.**
  - $C_3 : (P_2 \lor \neg P_3)$ = $\text{True} \lor \text{False}$ = **True.**
  - $C_4 : (\neg P_2 \lor P_3)$ = $\text{False} \lor \text{True}$ = **True.**



Analysis and Implication:

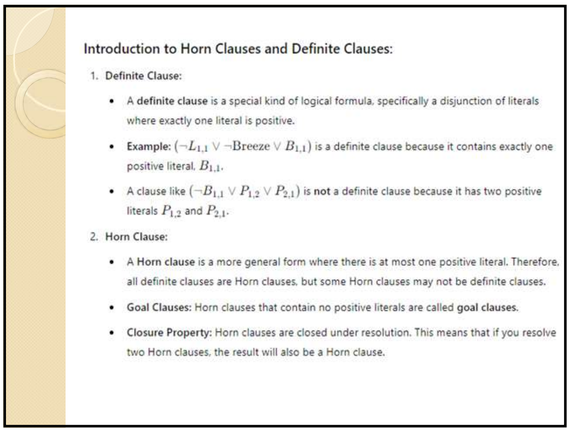
1. **No Contradiction:** The constructed model satisfies all clauses in S. There is no point where a clause is falsified.
2. **Satisfiability:** Since we successfully constructed a model that satisfies S, the set of clauses S is **satisfiable.**
3. **Resolution Completeness:** If the resolution closure $RC(S)$ had led to the empty clause, S would have been unsatisfiable. But in this case, since we found a model, S is satisfiable, confirming that the resolution process is indeed complete: it would have found a contradiction if one existed.

- The model $\{P_1 = \text{True}, P_2 = \text{True}, P_3 = \text{True}\}$ satisfies all the clauses.
- This example illustrates how constructing a model verifies the satisfiability of the clause set, and shows the completeness of the resolution method: it finds a contradiction if and only if the clause set is unsatisfiable.

Example Clauses:

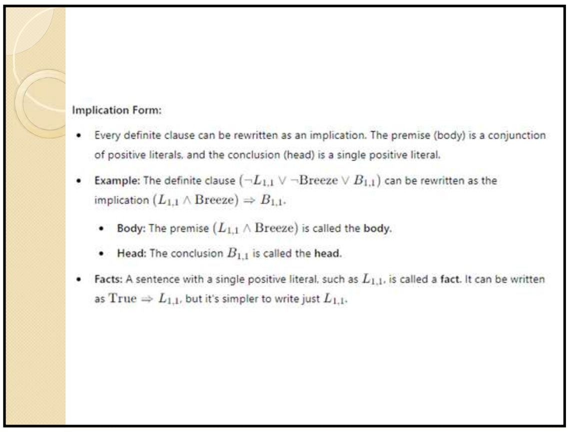
Consider the following set S of clauses:

1. $C_1 : (P_1)$
2. $C_2 : (\neg P_1)$
3. $C_3 : (P_2 \vee P_3)$
4. $C_4 : (\neg P_2)$
5. $C_5 : (\neg P_3)$



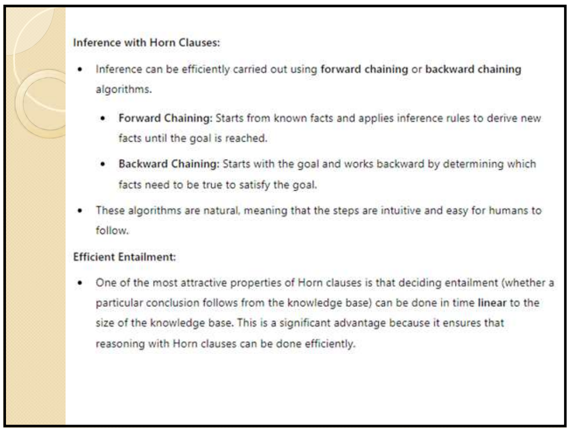
Introduction to Horn Clauses and Definite Clauses:

1. **Definite Clause:**
   - A **definite clause** is a special kind of logical formula, specifically a disjunction of literals where exactly one literal is positive.
   - **Example:** $(\neg L_{1,1} \vee \neg \text{Breeze} \vee B_{1,1})$ is a definite clause because it contains exactly one positive literal, $B_{1,1}$.
   - A clause like $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ is **not** a definite clause because it has two positive literals $P_{1,2}$ and $P_{2,1}$.
2. **Horn Clause:**
   - A **Horn clause** is a more general form where there is at most one positive literal. Therefore, all definite clauses are Horn clauses, but some Horn clauses may not be definite clauses.
   - **Goal Clauses:** Horn clauses that contain no positive literals are called **goal clauses.**
   - **Closure Property:** Horn clauses are closed under resolution. This means that if you resolve two Horn clauses, the result will also be a Horn clause.



Implication Form:

- Every definite clause can be rewritten as an implication. The premise (body) is a conjunction of positive literals, and the conclusion (head) is a single positive literal.
- **Example:** The definite clause $(\neg L_{1,1} \vee \neg \text{Breeze} \vee B_{1,1})$ can be rewritten as the implication $(L_{1,1} \wedge \text{Breeze}) \Rightarrow B_{1,1}$.
  - **Body:** The premise $(L_{1,1} \wedge \text{Breeze})$ is called the **body.**
  - **Head:** The conclusion $B_{1,1}$ is called the **head.**
- **Facts:** A sentence with a single positive literal, such as $L_{1,1}$, is called a **fact.** It can be written as $\text{True} \Rightarrow L_{1,1}$, but it's simpler to write just $L_{1,1}$.

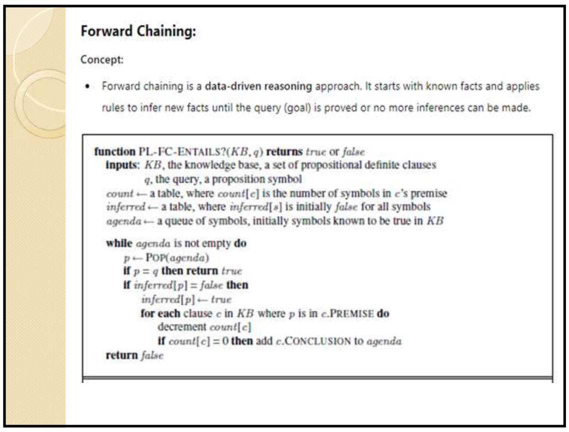


Inference with Horn Clauses:

- Inference can be efficiently carried out using **forward chaining** or **backward chaining** algorithms.
  - **Forward Chaining:** Starts from known facts and applies inference rules to derive new facts until the goal is reached.
  - **Backward Chaining:** Starts with the goal and works backward by determining which facts need to be true to satisfy the goal.
- These algorithms are natural, meaning that the steps are intuitive and easy for humans to follow.

Efficient Entailment:

- One of the most attractive properties of Horn clauses is that deciding entailment (whether a particular conclusion follows from the knowledge base) can be done in time **linear** to the size of the knowledge base. This is a significant advantage because it ensures that reasoning with Horn clauses can be done efficiently.

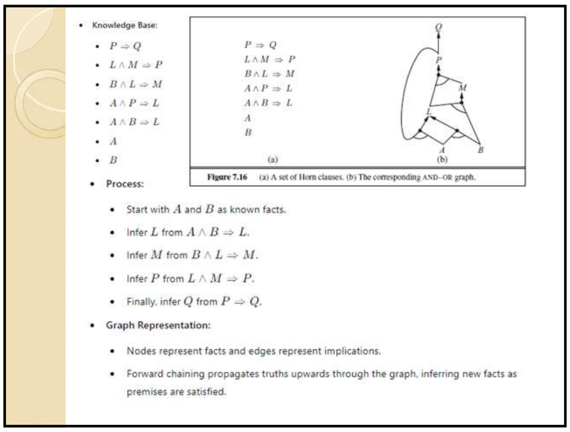


Forward Chaining:

Concept:

- Forward chaining is a **data-driven reasoning** approach. It starts with known facts and applies rules to infer new facts until the query (goal) is proved or no more inferences can be made.

```
function PL-FC-ENTAILS?($KB, q$) returns $true$ or $false$
  inputs: $KB$, the knowledge base, a set of propositional definite clauses
          $q$, the query, a proposition symbol
  $count$ ← a table, where $count[c]$ is the number of symbols in $c$'s premise
  $inferred$ ← a table, where $inferred[s]$ is initially $false$ for all symbols
  $agenda$ ← a queue of symbols, initially symbols known to be true in $KB$

  while $agenda$ is not empty do
      $p$ ← POP($agenda$)
      if $p = q$ then return $true$
      if $inferred[p] = false$ then
          $inferred[p]$ ← $true$
          for each clause $c$ in $KB$ where $p$ is in $c$.PREMISE do
              decrement $count[c]$
              if $count[c] = 0$ then add $c$.CONCLUSION to $agenda$
  return $false$
```

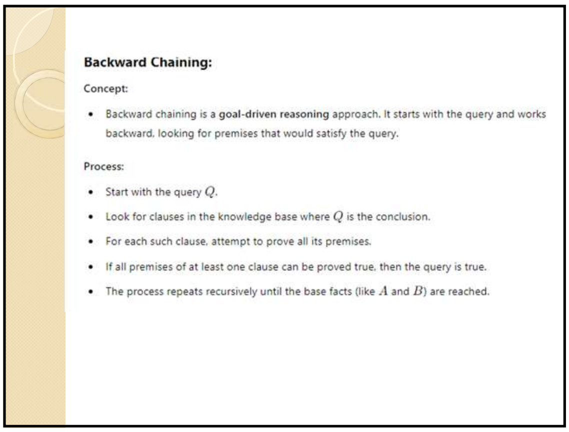


- Knowledge Base:
  - $P \Rightarrow Q$
  - $L \wedge M \Rightarrow P$
  - $B \wedge L \Rightarrow M$
  - $A \wedge P \Rightarrow L$
  - $A \wedge B \Rightarrow L$
  - A
  - B

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B
(a)
Q
P
M
L
A
B
(b)

Figure 7.16 (a) A set of Horn clauses. (b) The corresponding AND–OR graph.

- Process:
  - Start with A and B as known facts.
  - Infer L from $A \wedge B \Rightarrow L$.
  - Infer M from $B \wedge L \Rightarrow M$.
  - Infer P from $L \wedge M \Rightarrow P$.
  - Finally, infer Q from $P \Rightarrow Q$.
- Graph Representation:
  - Nodes represent facts and edges represent implications.
  - Forward chaining propagates truths upwards through the graph, inferring new facts as premises are satisfied.



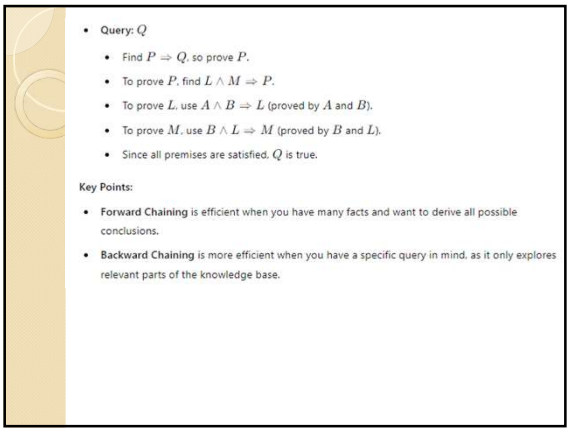
Backward Chaining:

Concept:

- Backward chaining is a **goal-driven reasoning** approach. It starts with the query and works backward, looking for premises that would satisfy the query.

Process:

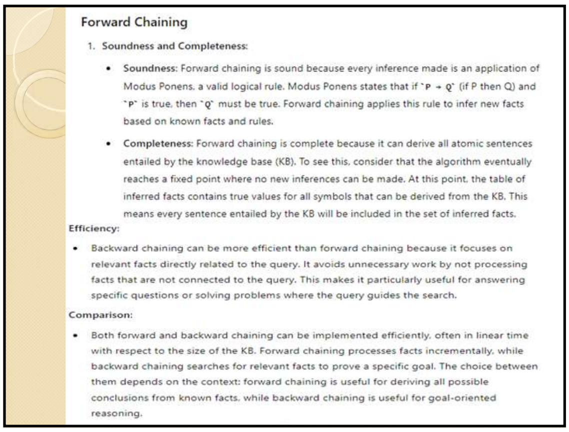
- Start with the query Q.
- Look for clauses in the knowledge base where Q is the conclusion.
- For each such clause, attempt to prove all its premises.
- If all premises of at least one clause can be proved true, then the query is true.
- The process repeats recursively until the base facts (like A and B) are reached.



- **Query:** Q
  - Find $P \Rightarrow Q$, so prove P.
  - To prove P, find $L \wedge M \Rightarrow P$.
  - To prove L, use $A \wedge B \Rightarrow L$ (proved by A and B).
  - To prove M, use $B \wedge L \Rightarrow M$ (proved by B and L).
  - Since all premises are satisfied, Q is true.

Key Points:

- **Forward Chaining** is efficient when you have many facts and want to derive all possible conclusions.
- **Backward Chaining** is more efficient when you have a specific query in mind, as it only explores relevant parts of the knowledge base.



Forward Chaining

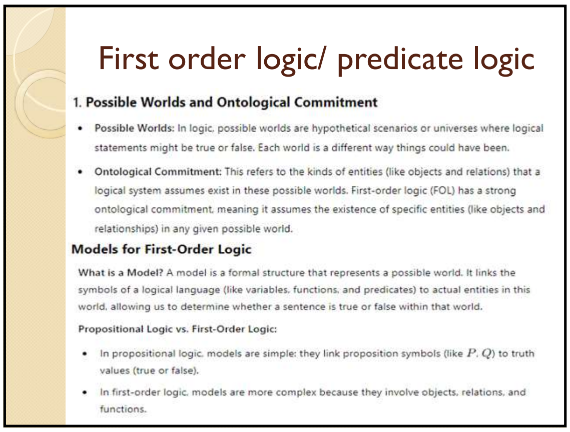
1. **Soundness and Completeness:**
   - **Soundness:** Forward chaining is sound because every inference made is an application of Modus Ponens, a valid logical rule. Modus Ponens states that if `P → Q` (if P then Q) and `P` is true, then `Q` must be true. Forward chaining applies this rule to infer new facts based on known facts and rules.
   - **Completeness:** Forward chaining is complete because it can derive all atomic sentences entailed by the knowledge base (KB). To see this, consider that the algorithm eventually reaches a fixed point where no new inferences can be made. At this point, the table of inferred facts contains true values for all symbols that can be derived from the KB. This means every sentence entailed by the KB will be included in the set of inferred facts.

Efficiency:

- Backward chaining can be more efficient than forward chaining because it focuses on relevant facts directly related to the query. It avoids unnecessary work by not processing facts that are not connected to the query. This makes it particularly useful for answering specific questions or solving problems where the query guides the search.

Comparison:

- Both forward and backward chaining can be implemented efficiently, often in linear time with respect to the size of the KB. Forward chaining processes facts incrementally, while backward chaining searches for relevant facts to prove a specific goal. The choice between them depends on the context: forward chaining is useful for deriving all possible conclusions from known facts, while backward chaining is useful for goal-oriented reasoning.



First order logic/ predicate logic

1. Possible Worlds and Ontological Commitment

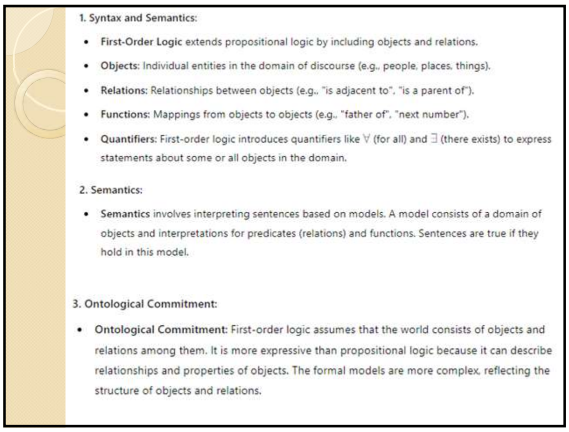
- **Possible Worlds:** In logic, possible worlds are hypothetical scenarios or universes where logical statements might be true or false. Each world is a different way things could have been.
- **Ontological Commitment:** This refers to the kinds of entities (like objects and relations) that a logical system assumes exist in these possible worlds. First-order logic (FOL) has a strong ontological commitment, meaning it assumes the existence of specific entities (like objects and relationships) in any given possible world.

Models for First-Order Logic

What is a Model? A model is a formal structure that represents a possible world. It links the symbols of a logical language (like variables, functions, and predicates) to actual entities in this world, allowing us to determine whether a sentence is true or false within that world.

Propositional Logic vs. First-Order Logic:

- In propositional logic, models are simple: they link proposition symbols (like P, Q) to truth values (true or false).
- In first-order logic, models are more complex because they involve objects, relations, and functions.



1. Syntax and Semantics:

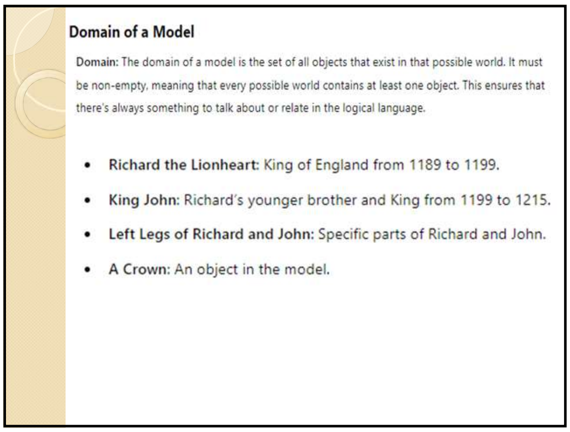
- **First-Order Logic** extends propositional logic by including objects and relations.
- **Objects:** Individual entities in the domain of discourse (e.g., people, places, things).
- **Relations:** Relationships between objects (e.g., "is adjacent to", "is a parent of").
- **Functions:** Mappings from objects to objects (e.g., "father of", "next number").
- **Quantifiers:** First-order logic introduces quantifiers like $\forall$ (for all) and $\exists$ (there exists) to express statements about some or all objects in the domain.

2. Semantics:

- **Semantics** involves interpreting sentences based on models. A model consists of a domain of objects and interpretations for predicates (relations) and functions. Sentences are true if they hold in this model.

3. Ontological Commitment:

- **Ontological Commitment:** First-order logic assumes that the world consists of objects and relations among them. It is more expressive than propositional logic because it can describe relationships and properties of objects. The formal models are more complex, reflecting the structure of objects and relations.



Domain of a Model

Domain: The domain of a model is the set of all objects that exist in that possible world. It must be non-empty, meaning that every possible world contains at least one object. This ensures that there's always something to talk about or relate in the logical language.

- **Richard the Lionheart:** King of England from 1189 to 1199.
- **King John:** Richard's younger brother and King from 1199 to 1215.
- **Left Legs of Richard and John:** Specific parts of Richard and John.
- **A Crown:** An object in the model.

Relations in the Model

What is a Relation? A relation links objects in the model. Formally, it is a set of ordered pairs (or tuples) of objects.

Unary Relations (Properties)

What is a Unary Relation? A unary relation is a property that applies to a single object.

Functions in the Model

What is a Function? A function is a special type of relation where each object is related to exactly one object.

Total Functions: In first-order logic, all functions must be total. This means that every possible input (like every object in the domain) must have a corresponding output.

Link Between Model Elements and Logical Sentences

Finally, the model must connect the objects, relations, and functions to the logical vocabulary used in sentences. This connection allows us to interpret sentences and determine their truth value within the context of the model.

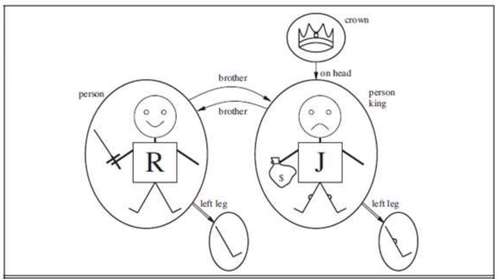


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

Objects:

1. **R (Richard the Lionheart):** Represented as a stick figure labeled "R."
2. **J (King John):** Represented as a stick figure labeled "J."
3. **Crown:** An object placed above John's head.
4. **Left leg of Richard:** Shown as an oval connected to Richard.
5. **Left leg of John:** Shown as an oval connected to John.

Binary Relations: { ⟨Richard the Lionheart, King John⟩, ⟨King John, Richard the Lionheart⟩ }

1. **Brother Relation:** Represented by an arrow labeled "brother" pointing from Richard (R) to John (J) and another arrow from John (J) to Richard (R). This shows that Richard and John are brothers, and this relationship is bidirectional.
2. **On Head Relation:** Represented by an arrow labeled "on head" pointing from the crown to John (J). This indicates that the crown is on King John's head. ⟨the crown, King John⟩.

Unary Relations (Properties):

1. **Person:** Labeled on both Richard (R) and John (J), indicating that both are persons.
2. **King:** Labeled only on John (J), indicating that John is a king (Richard is not labeled as a king, possibly because he is considered dead in this model).
3. **Crown:** Labeled on the crown, indicating that this object is a crown.

Unary Function:

1. **Left Leg Function:**
   - An arrow labeled "left leg" points from Richard (R) to his left leg and from John (J) to his left leg. This indicates that the function maps each person to their respective left leg.
     - There's a function mapping each person to their left leg:
       - Richard the Lionheart → Richard's left leg
       - King John → John's left leg

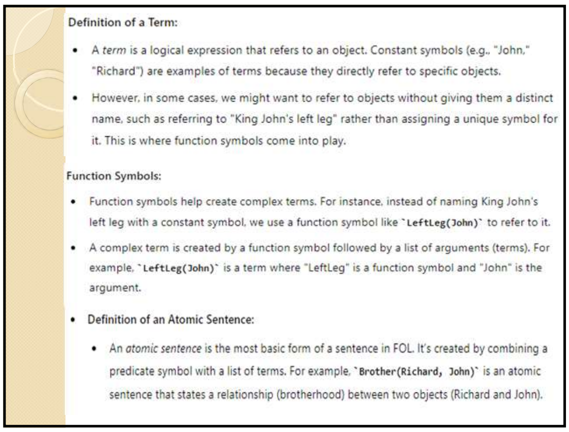
Summary

This model illustrates how first-order logic can represent a possible world with specific objects and relationships:

- Richard and John are both persons.
- John is a king, but Richard is not, possibly because he is dead at this point.
- Richard and John are brothers.
- The crown is on John's head.
- Each person has one left leg, which is specified by the "left leg" function.

```
        Sentence  →  AtomicSentence | ComplexSentence
  AtomicSentence  →  Predicate | Predicate(Term,...) | Term = Term
 ComplexSentence  →  ( Sentence ) | [ Sentence ]
                  |  ¬ Sentence
                  |  Sentence ∧ Sentence
                  |  Sentence ∨ Sentence
                  |  Sentence ⇒ Sentence
                  |  Sentence ⇔ Sentence
                  |  Quantifier Variable,... Sentence

            Term  →  Function(Term,...)
                  |  Constant
                  |  Variable

      Quantifier  →  ∀ | ∃
        Constant  →  A | X₁ | John | ···
        Variable  →  a | x | s | ···
       Predicate  →  True | False | After | Loves | Raining | ···
        Function  →  Mother | LeftLeg | ···
OPERATOR PRECEDENCE  :  ¬, =, ∧, ∨, ⇒, ⇔
```

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form

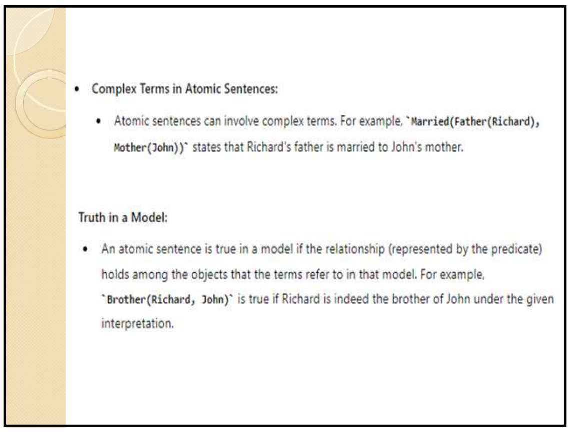


Definition of a Term:

- A *term* is a logical expression that refers to an object. Constant symbols (e.g., "John," "Richard") are examples of terms because they directly refer to specific objects.
- However, in some cases, we might want to refer to objects without giving them a distinct name, such as referring to "King John's left leg" rather than assigning a unique symbol for it. This is where function symbols come into play.

Function Symbols:

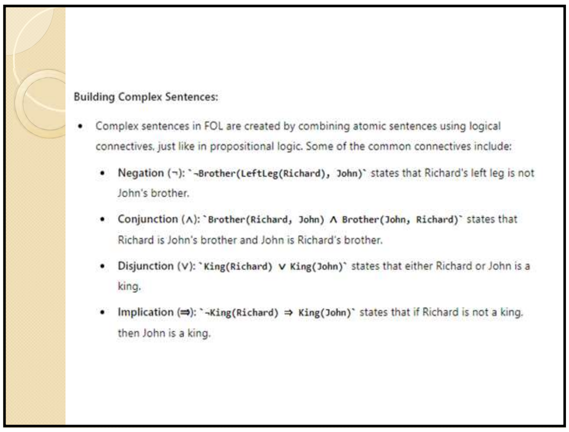
- Function symbols help create complex terms. For instance, instead of naming King John's left leg with a constant symbol, we use a function symbol like `LeftLeg(John)` to refer to it.
- A complex term is created by a function symbol followed by a list of arguments (terms). For example, `LeftLeg(John)` is a term where "LeftLeg" is a function symbol and "John" is the argument.
- **Definition of an Atomic Sentence:**
  - An *atomic sentence* is the most basic form of a sentence in FOL. It's created by combining a predicate symbol with a list of terms. For example, `Brother(Richard, John)` is an atomic sentence that states a relationship (brotherhood) between two objects (Richard and John).



- **Complex Terms in Atomic Sentences:**
  - Atomic sentences can involve complex terms. For example, `Married(Father(Richard), Mother(John))` states that Richard's father is married to John's mother.

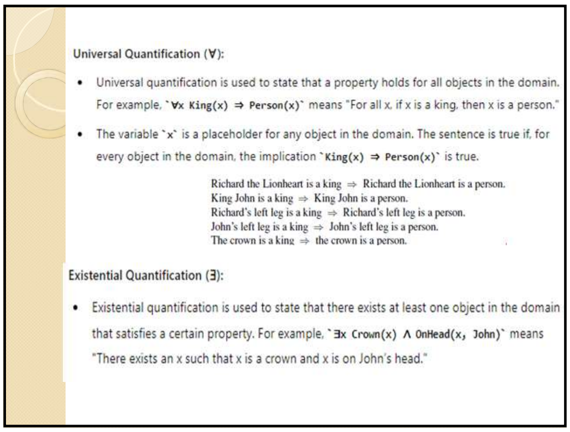
Truth in a Model:

- An atomic sentence is true in a model if the relationship (represented by the predicate) holds among the objects that the terms refer to in that model. For example, `Brother(Richard, John)` is true if Richard is indeed the brother of John under the given interpretation.



Building Complex Sentences:

- Complex sentences in FOL are created by combining atomic sentences using logical connectives, just like in propositional logic. Some of the common connectives include:
  - **Negation (¬):** `¬Brother(LeftLeg(Richard), John)` states that Richard's left leg is not John's brother.
  - **Conjunction (∧):** `Brother(Richard, John) ∧ Brother(John, Richard)` states that Richard is John's brother and John is Richard's brother.
  - **Disjunction (∨):** `King(Richard) ∨ King(John)` states that either Richard or John is a king.
  - **Implication (⇒):** `¬King(Richard) ⇒ King(John)` states that if Richard is not a king, then John is a king.



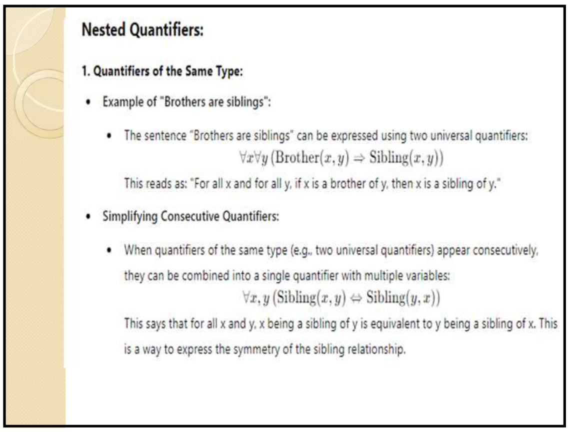
Universal Quantification (∀):

- Universal quantification is used to state that a property holds for all objects in the domain. For example, `∀x King(x) ⇒ Person(x)` means "For all x, if x is a king, then x is a person."
- The variable `x` is a placeholder for any object in the domain. The sentence is true if, for every object in the domain, the implication `King(x) ⇒ Person(x)` is true.

Richard the Lionheart is a king ⇒ Richard the Lionheart is a person.
King John is a king ⇒ King John is a person.
Richard's left leg is a king ⇒ Richard's left leg is a person.
John's left leg is a king ⇒ John's left leg is a person.
The crown is a king ⇒ the crown is a person.

Existential Quantification (∃):

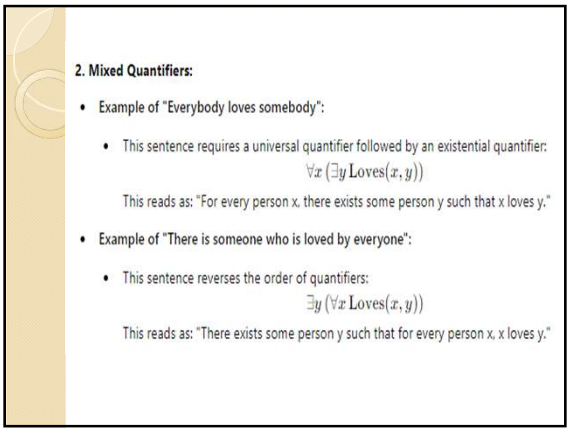
- Existential quantification is used to state that there exists at least one object in the domain that satisfies a certain property. For example, `∃x Crown(x) ∧ OnHead(x, John)` means "There exists an x such that x is a crown and x is on John's head."



Nested Quantifiers:

1. Quantifiers of the Same Type:

- **Example of "Brothers are siblings":**
  - The sentence "Brothers are siblings" can be expressed using two universal quantifiers:
    $$\forall x \forall y\, (\text{Brother}(x, y) \Rightarrow \text{Sibling}(x, y))$$
    This reads as: "For all x and for all y, if x is a brother of y, then x is a sibling of y."
- **Simplifying Consecutive Quantifiers:**
  - When quantifiers of the same type (e.g., two universal quantifiers) appear consecutively, they can be combined into a single quantifier with multiple variables:
    $$\forall x, y\, (\text{Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x))$$
    This says that for all x and y, x being a sibling of y is equivalent to y being a sibling of x. This is a way to express the symmetry of the sibling relationship.



2. Mixed Quantifiers:

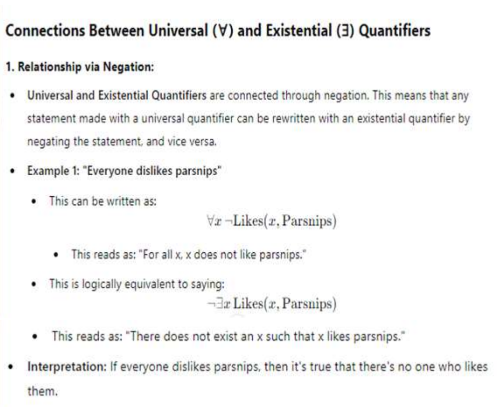
- **Example of "Everybody loves somebody":**
  - This sentence requires a universal quantifier followed by an existential quantifier:
    $$\forall x\, (\exists y\, \text{Loves}(x, y))$$
    This reads as: "For every person x, there exists some person y such that x loves y."
- **Example of "There is someone who is loved by everyone":**
  - This sentence reverses the order of quantifiers:
    $$\exists y\, (\forall x\, \text{Loves}(x, y))$$
    This reads as: "There exists some person y such that for every person x, x loves y."

Importance of Quantifier Order

- The order of quantifiers matters significantly because it changes the meaning of the sentence.
  - **∀x (∃y {Loves}(x, y))** means everyone has at least one person they love.
  - **∃y (∀x {Loves}(x, y))** means there is a specific person who is loved by everyone.
- Parentheses can help clarify the scope of each quantifier:
  - **∀x (∃y {Loves}(x, y))** emphasizes that for each x, there exists some y.
  - **∃y (∀x {Loves}(x, y))** emphasizes that there is a particular y who is loved by all x's.

Avoiding Confusion with Variable Names

- **Same Variable Name Used Twice:**
  - Consider the sentence: ∀x(Crown(x)∨(∃xBrother(Richard,x)))
  - This sentence uses the variable name "x" in both the universal and existential quantifiers.
- To avoid confusion, it's recommended to use different variable names when working with nested quantifiers:
- Instead of **∃x {Brother}(Richard, x)** inside **∀x**,
  - use **∃z {Brother}(Richard, z)}** to clarify that the inner quantifier refers to a different object.



Connections Between Universal (∀) and Existential (∃) Quantifiers

1. Relationship via Negation:

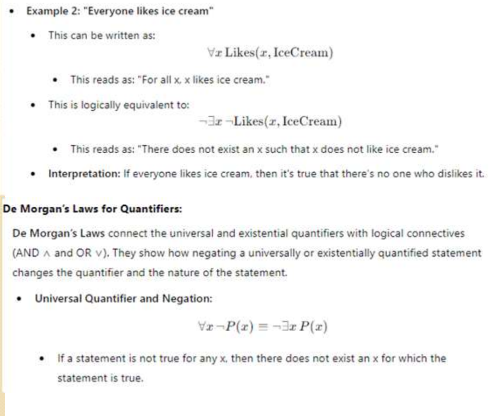
- **Universal and Existential Quantifiers** are connected through negation. This means that any statement made with a universal quantifier can be rewritten with an existential quantifier by negating the statement, and vice versa.
- **Example 1: "Everyone dislikes parsnips"**
  - This can be written as:

$$\forall x\, \neg \text{Likes}(x, \text{Parsnips})$$

    - This reads as: "For all x, x does not like parsnips."
  - This is logically equivalent to saying:

$$\neg \exists x\, \text{Likes}(x, \text{Parsnips})$$

  - This reads as: "There does not exist an x such that x likes parsnips."
- **Interpretation:** If everyone dislikes parsnips, then it's true that there's no one who likes them.



- **Example 2: "Everyone likes ice cream"**
  - This can be written as:

$$\forall x\, \text{Likes}(x, \text{IceCream})$$

    - This reads as: "For all x, x likes ice cream."
  - This is logically equivalent to:

$$\neg \exists x\, \neg \text{Likes}(x, \text{IceCream})$$

    - This reads as: "There does not exist an x such that x does not like ice cream."
  - **Interpretation:** If everyone likes ice cream, then it's true that there's no one who dislikes it.

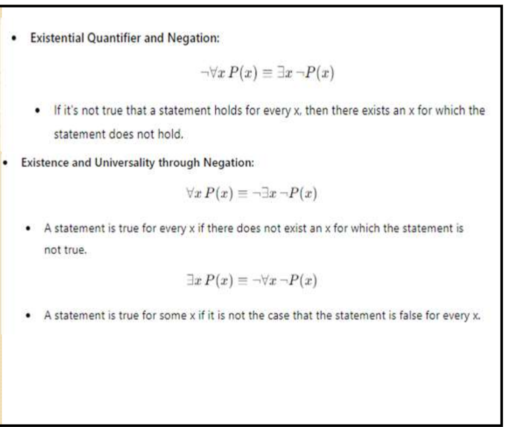
De Morgan's Laws for Quantifiers:

De Morgan's Laws connect the universal and existential quantifiers with logical connectives (AND ∧ and OR ∨). They show how negating a universally or existentially quantified statement changes the quantifier and the nature of the statement.

- **Universal Quantifier and Negation:**

$$\forall x\, \neg P(x) \equiv \neg \exists x\, P(x)$$

  - If a statement is not true for any x, then there does not exist an x for which the statement is true.



- **Existential Quantifier and Negation:**

$$\neg \forall x\, P(x) \equiv \exists x\, \neg P(x)$$

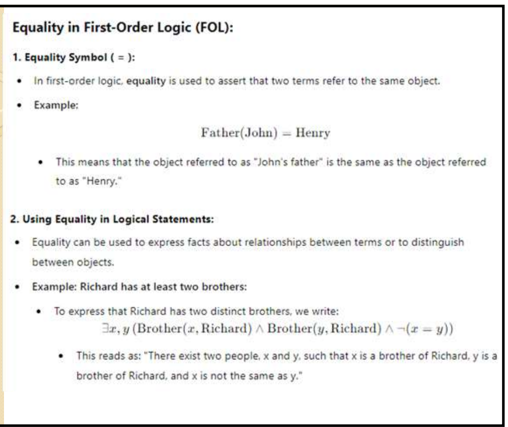
  - If it's not true that a statement holds for every x, then there exists an x for which the statement does not hold.
- **Existence and Universality through Negation:**

$$\forall x\, P(x) \equiv \neg \exists x\, \neg P(x)$$

  - A statement is true for every x if there does not exist an x for which the statement is not true.

$$\exists x\, P(x) \equiv \neg \forall x\, \neg P(x)$$

  - A statement is true for some x if it is not the case that the statement is false for every x.



Equality in First-Order Logic (FOL):

1. Equality Symbol (=):

- In first-order logic, **equality** is used to assert that two terms refer to the same object.
- **Example:**

$$\text{Father}(\text{John}) = \text{Henry}$$

  - This means that the object referred to as "John's father" is the same as the object referred to as "Henry."

2. Using Equality in Logical Statements:

- Equality can be used to express facts about relationships between terms or to distinguish between objects.
- **Example: Richard has at least two brothers:**
  - To express that Richard has two distinct brothers, we write:

$$\exists x, y\, (\text{Brother}(x, \text{Richard}) \land \text{Brother}(y, \text{Richard}) \land \neg(x = y))$$

    - This reads as: "There exist two people, x and y, such that x is a brother of Richard, y is a brother of Richard, and x is not the same as y."

PL Vs FOL

1. Inference rules for quantifiers

Universal Instantiation (UI):

1. **Definition:**
   - Universal Instantiation is a rule that allows us to derive specific instances from a universally quantified statement. If something is true for all elements in a domain, then it is true for any particular element in that domain.
2. **Example with Universal Quantifiers:**

$$\frac{\forall v \; \alpha}{\textsc{Subst}(\{v/g\}, \alpha)}$$

   - Consider the axiom:

$$\forall x \, (King(x) \wedge Greedy(x) \Rightarrow Evil(x))$$

This statement means that "for all x, if x is a king and x is greedy, then x is evil." Using Universal Instantiation, we can derive specific instances:

- If John is a king and greedy, then John is evil:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$

- If Richard is a king and greedy, then Richard is evil:

$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$

- If the father of John is a king and greedy, then the father of John is evil:

$$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John))$$

Where "v" is a variable and "g" is a ground term. The specific sentences given above are obtained using the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

Existential Instantiation (EI):

1. **Definition:**
   - Existential Instantiation is a rule that allows us to take a statement that asserts the existence of some object with a certain property and replace the existentially quantified variable with a new constant symbol that doesn't appear elsewhere in the knowledge base. This process gives a name to the unknown object.
2. **Example with Existential Quantifiers:**
   - Consider the sentence:

$$\exists x \, (Crown(x) \wedge OnHead(x, John))$$

This states that "there exists an x such that x is a crown and x is on John's head." Using Existential Instantiation, we can infer:

$$Crown(C1) \wedge OnHead(C1, John)$$

Here, "C1" is a new constant symbol representing some specific crown that satisfies the condition.

Skolemization:

- The process of introducing a new constant symbol (like "C1") is a special case of a more general process called **skolemization**. The new name or constant introduced must not already belong to another object in the knowledge base.

Difference between UI and EI:

- **Universal Instantiation** can be applied multiple times, as it deals with a universally quantified sentence that can lead to multiple specific instances.
- **Existential Instantiation** is typically applied only once because the existentially quantified sentence is satisfied by finding just one instance. Once the sentence is instantiated, the original existential statement can be discarded.

After applying Existential Instantiation, the new knowledge base is not logically equivalent to the original knowledge base, but it is **inferentially equivalent.**

Reduction to Propositional Inference

1. **Basic Concept:**
   - The idea is that once we have rules to derive non-quantified (i.e., propositional) sentences from quantified sentences, we can reduce the problem of inference in first-order logic (FOL) to a problem of inference in propositional logic, which is simpler and better understood.
2. **Replacement of Quantified Sentences:**
   - **Existential Quantifiers:**
     - In FOL, an existentially quantified sentence (e.g., "∃x P(x)") can be replaced by a single instantiation (e.g., "P(c)," where "c" is a specific constant).
   - **Universal Quantifiers:**
     - A universally quantified sentence (e.g., "∀x P(x)") can be replaced by the set of all possible instantiations with ground terms from the knowledge base. This means that for each possible value of the variable "x," you create a specific sentence.

Example of Propositionalization:

- Consider a knowledge base with the following sentences:
  - $\forall x (King(x) \wedge Greedy(x) \Rightarrow Evil(x))$
  - $King(John)$
  - $Greedy(John)$
  - $Brother(Richard, John)$
- To apply propositionalization:
  - Use **Universal Instantiation (UI)** on the universally quantified sentence with all possible ground-term substitutions, such as $\{x/John\}$ and $\{x/Richard\}$.
  - This gives:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$

$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$

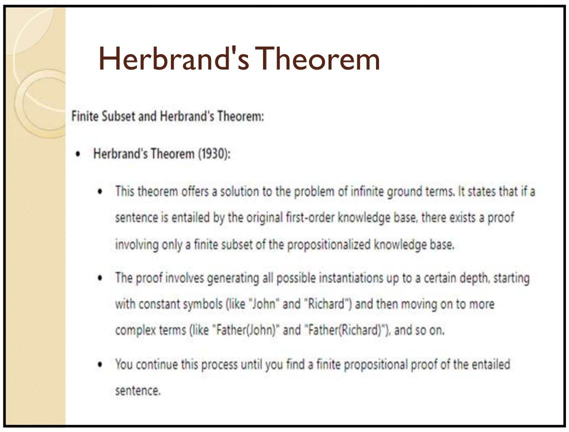
  - The original universally quantified sentence $\forall x (King(x) \wedge Greedy(x) \Rightarrow Evil(x))$ is then discarded because we've covered all relevant ground terms.

Complete Propositionalization:

- Propositionalization can be generalized to any first-order knowledge base and query. This means that the process can convert any first-order logic problem into a propositional logic problem while preserving **entailment** (i.e., the logical consequences of the knowledge base remain the same).

The Problem with Function Symbols:

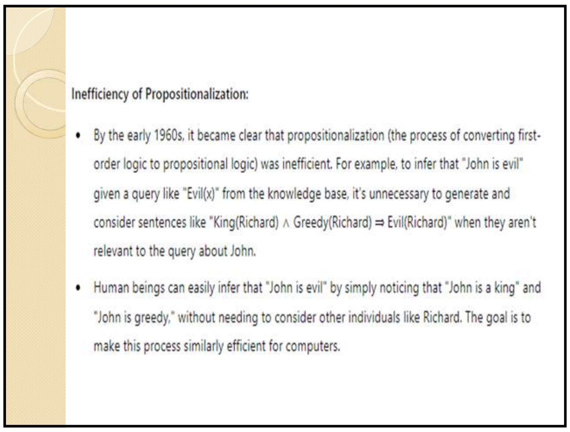
- A significant challenge arises when the knowledge base includes function symbols (e.g., "Father"). Function symbols can generate an infinite number of ground terms. For instance, if "Father" is a function symbol, you could have an infinite sequence of terms like "Father(John)," "Father(Father(John))," "Father(Father(Father(John)))," etc.
- The set of possible ground-term substitutions becomes infinite, leading to an infinitely large propositional knowledge base, which traditional propositional algorithms cannot handle.



Herbrand's Theorem

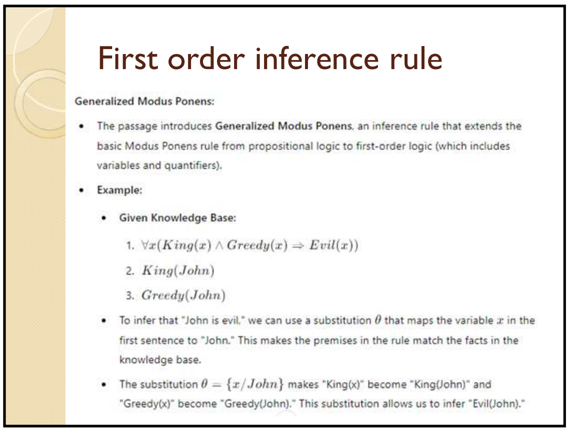
Finite Subset and Herbrand's Theorem:

- **Herbrand's Theorem (1930):**
  - This theorem offers a solution to the problem of infinite ground terms. It states that if a sentence is entailed by the original first-order knowledge base, there exists a proof involving only a finite subset of the propositionalized knowledge base.
  - The proof involves generating all possible instantiations up to a certain depth, starting with constant symbols (like "John" and "Richard") and then moving on to more complex terms (like "Father(John)" and "Father(Richard)"), and so on.
  - You continue this process until you find a finite propositional proof of the entailed sentence.



Inefficiency of Propositionalization:

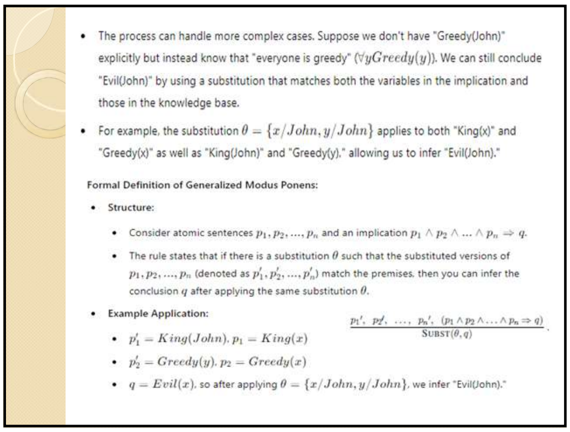
- By the early 1960s, it became clear that propositionalization (the process of converting first-order logic to propositional logic) was inefficient. For example, to infer that "John is evil" given a query like "Evil(x)" from the knowledge base, it's unnecessary to generate and consider sentences like "King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)" when they aren't relevant to the query about John.
- Human beings can easily infer that "John is evil" by simply noticing that "John is a king" and "John is greedy," without needing to consider other individuals like Richard. The goal is to make this process similarly efficient for computers.



First order inference rule

Generalized Modus Ponens:

- The passage introduces **Generalized Modus Ponens**, an inference rule that extends the basic Modus Ponens rule from propositional logic to first-order logic (which includes variables and quantifiers).
- **Example:**
  - **Given Knowledge Base:**
    1. $\forall x(King(x) \land Greedy(x) \Rightarrow Evil(x))$
    2. $King(John)$
    3. $Greedy(John)$
  - To infer that "John is evil," we can use a substitution θ that maps the variable x in the first sentence to "John." This makes the premises in the rule match the facts in the knowledge base.
  - The substitution $\theta = \{x/John\}$ makes "King(x)" become "King(John)" and "Greedy(x)" become "Greedy(John)." This substitution allows us to infer "Evil(John)."



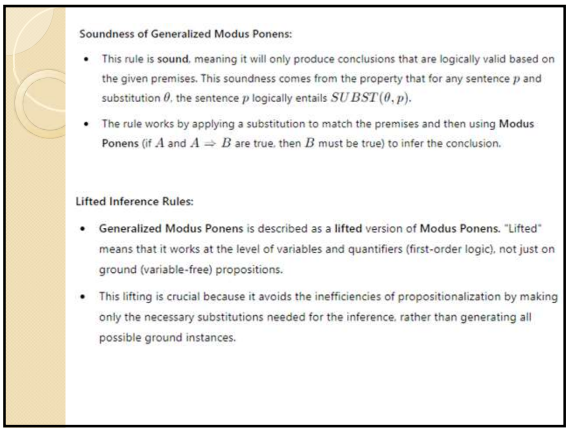
- The process can handle more complex cases. Suppose we don't have "Greedy(John)" explicitly but instead know that "everyone is greedy" ($\forall y Greedy(y)$). We can still conclude "Evil(John)" by using a substitution that matches both the variables in the implication and those in the knowledge base.
- For example, the substitution $\theta = \{x/John, y/John\}$ applies to both "King(x)" and "Greedy(x)" as well as "King(John)" and "Greedy(y)," allowing us to infer "Evil(John)."

Formal Definition of Generalized Modus Ponens:

- **Structure:**
  - Consider atomic sentences $p_1, p_2, ..., p_n$ and an implication $p_1 \land p_2 \land ... \land p_n \Rightarrow q$.
  - The rule states that if there is a substitution θ such that the substituted versions of $p_1, p_2, ..., p_n$ (denoted as $p'_1, p'_2, ..., p'_n$) match the premises, then you can infer the conclusion q after applying the same substitution θ.
- **Example Application:**

$$\frac{p_1{}',\ p_2{}',\ \ldots,\ p_n{}',\ (p_1 \land p_2 \land \ldots \land p_n \Rightarrow q)}{\textsc{Subst}(\theta, q)}.$$

  - $p'_1 = King(John)$, $p_1 = King(x)$
  - $p'_2 = Greedy(y)$, $p_2 = Greedy(x)$
  - $q = Evil(x)$, so after applying $\theta = \{x/John, y/John\}$, we infer "Evil(John)."

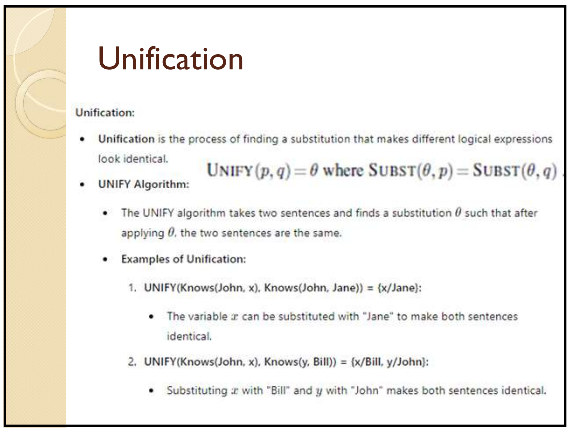


Soundness of Generalized Modus Ponens:

- This rule is **sound**, meaning it will only produce conclusions that are logically valid based on the given premises. This soundness comes from the property that for any sentence p and substitution θ, the sentence p logically entails $SUBST(\theta, p)$.
- The rule works by applying a substitution to match the premises and then using **Modus Ponens** (if A and $A \Rightarrow B$ are true, then B must be true) to infer the conclusion.

Lifted Inference Rules:

- **Generalized Modus Ponens** is described as a **lifted** version of **Modus Ponens.** "Lifted" means that it works at the level of variables and quantifiers (first-order logic), not just on ground (variable-free) propositions.
- This lifting is crucial because it avoids the inefficiencies of propositionalization by making only the necessary substitutions needed for the inference, rather than generating all possible ground instances.



Unification

Unification:

- **Unification** is the process of finding a substitution that makes different logical expressions look identical.

$$\textsc{Unify}(p, q) = \theta \text{ where } \textsc{Subst}(\theta, p) = \textsc{Subst}(\theta, q)$$

- **UNIFY Algorithm:**
  - The UNIFY algorithm takes two sentences and finds a substitution θ such that after applying θ, the two sentences are the same.
  - **Examples of Unification:**
    1. **UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane}:**
       - The variable x can be substituted with "Jane" to make both sentences identical.
    2. **UNIFY(Knows(John, x), Knows(y, Bill)) = {x/Bill, y/John}:**
       - Substituting x with "Bill" and y with "John" makes both sentences identical.

3. UNIFY(Knows(John, x), Knows(y, Mother(y))) = {y/John, x/Mother(John)}:
   - Substituting y with "John" and x with "Mother(John)" makes the sentences identical.
4. UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail:
   - This unification fails because x cannot be both "John" and "Elizabeth" simultaneously.

- Problem:
  - Sometimes unification fails because of a variable name clash (e.g., both sentences use the same variable x, leading to confusion).
- Solution:
  - To avoid this, we can **standardize apart** one of the sentences by renaming its variables to avoid clashes. For example, rename x in one sentence to x_{17} (a new unique variable name). This allows unification to proceed correctly.

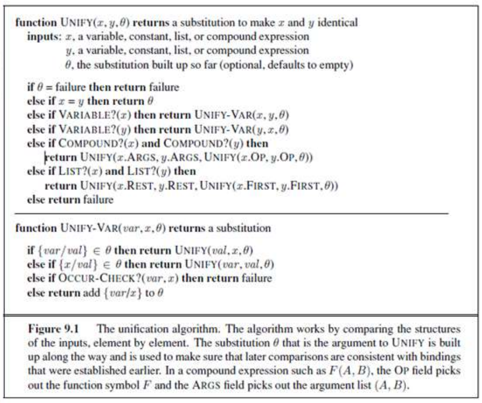
Example

- $\text{UNIFY}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(John, x)$	$Knows(John, Jane)$	$\{x/Jane\}$
$Knows(John, x)$	$Knows(y, Mary)$	$\{x/Mary, y/John\}$
$Knows(John, x)$	$Knows(y, Mother(y))$	$\{y/John, x/Mother(John)\}$
$Knows(John, x)$	$Knows(x, Mary)$	$fail$

- Standardizing apart eliminates overlap of variables, e.g., $Knows(z_{17}, Mary)$

$Knows(John, x)$	$Knows(z_{17}, Mary)$	$\{z_{17}/John, x/Mary\}$



```
function UNIFY($x, y, \theta$) returns a substitution to make $x$ and $y$ identical
  inputs: $x$, a variable, constant, list, or compound expression
          $y$, a variable, constant, list, or compound expression
          $\theta$, the substitution built up so far (optional, defaults to empty)

  if $\theta$ = failure then return failure
  else if $x = y$ then return $\theta$
  else if VARIABLE?($x$) then return UNIFY-VAR($x, y, \theta$)
  else if VARIABLE?($y$) then return UNIFY-VAR($y, x, \theta$)
  else if COMPOUND?($x$) and COMPOUND?($y$) then
     return UNIFY($x$.ARGS, $y$.ARGS, UNIFY($x$.OP, $y$.OP, $\theta$))
  else if LIST?($x$) and LIST?($y$) then
     return UNIFY($x$.REST, $y$.REST, UNIFY($x$.FIRST, $y$.FIRST, $\theta$))
  else return failure
```

```
function UNIFY-VAR($var, x, \theta$) returns a substitution

  if $\{var/val\} \in \theta$ then return UNIFY($val, x, \theta$)
  else if $\{x/val\} \in \theta$ then return UNIFY($var, val, \theta$)
  else if OCCUR-CHECK?($var, x$) then return failure
  else return add $\{var/x\}$ to $\theta$
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Most General Unifier (MGU):

- Multiple Unifiers:
  - Sometimes, there could be more than one way to unify two expressions. For example, $UNIFY(Knows(John, x), Knows(y, z))$ could return either $\{y/John, x/z\}$ or $\{y/John, x/John, z/John\}$.
- Generalization:
  - The first unifier is more general than the second because it places fewer restrictions on the variables' values. The **most general unifier (MGU)** is the most general substitution that can unify the two expressions. In this case, it is $\{y/John, x/z\}$.

UNIFY Algorithm and Occur Check:

- The UNIFY algorithm works by recursively comparing the structures of two expressions while building a unifier.
- Occur Check:
  - A crucial part of the algorithm is the **occur check**, which ensures that a variable does not appear within its own substitution term (e.g., $S(x)$ cannot unify with $S(S(x))$). This check is necessary to prevent constructing inconsistent or unsound unifiers.
  - However, the occur check makes the UNIFY algorithm quadratic in complexity, meaning it takes more time as the expressions get larger. Some systems skip the occur check to improve performance but risk making unsound inferences.

- **Generalized Modus Ponens** is a powerful inference rule that extends Modus Ponens to first-order logic, allowing computers to make inferences more efficiently without generating unnecessary ground instances.
- **Unification** is the process of finding substitutions that make different logical expressions identical. It's a core part of first-order inference.
- **Most General Unifier (MGU)** refers to the most general substitution that unifies two expressions, ensuring that the inference process remains as flexible as possible.
- **Occur Check** is a step in the UNIFY algorithm that prevents variables from being substituted in a way that would lead to logical inconsistencies.

Storage and Retrieval

1. Basic Operations: STORE and FETCH

- **STORE(s):** This function stores a sentence s into the knowledge base. It simply means adding new information or a fact into the database.
- **FETCH(q):** This function retrieves all unifiers such that the query q unifies with some sentence in the knowledge base. It's like asking the knowledge base to return all pieces of information that could match or answer the query.

2. Simplest Implementation of STORE and FETCH

- The simplest way to implement these operations is to keep all facts in a long list. When you need to retrieve information, FETCH simply unifies the query with every element in the list.
- **Efficiency Issue:** This method is inefficient because it involves checking each fact in the list, which can be time-consuming for large knowledge bases.

3. Predicate Indexing to Improve FETCH

- **Predicate Indexing:** To make FETCH more efficient, facts can be indexed by their predicate (the function name in the logical statement). For example, all facts related to the predicate "Knows" would be stored together, separate from facts related to the predicate "Brother."
- **Hash Table:** These indexed buckets can be stored in a hash table for fast retrieval, allowing the system to quickly access only those facts that are relevant to the query.

4. Challenges with Large Predicate Buckets

- **Example Scenario:** Suppose the knowledge base contains facts about employment relationships using the predicate `Employs(x, y)`. If there are millions of employers and employees, the `Employs` bucket would be enormous.
- **Query Efficiency:** A query like `Employs(x, Richard)` would require scanning the entire bucket, which is inefficient.

5. Multiple Index Keys for Better Querying

- To handle specific queries more efficiently, facts can be indexed using combined keys. For example, the system might index facts by both the predicate and the second argument, or the predicate and the first argument.
- **Example:** For the query `Employs(IBM, y)`, the system would use an index that combines the `Employs` predicate with the first argument (`IBM`), allowing it to directly retrieve the relevant facts.

6. Subsumption Lattices

- **Concept:** A subsumption lattice is a structure that organizes queries or logical sentences based on how they generalize or specify each other.
- **Example from Figure 9.2:**
  - **Figure 9.2(a):** Shows the subsumption lattice for the sentence `Employs(IBM, Richard)`.
    - **Lowest Node:** `Employs(IBM, Richard)` is the most specific query (asks for both the employer and employee).
    - **Intermediate Nodes:** `Employs(x, Richard)` and `Employs(IBM, y)` are more general queries.
    - **Highest Node:** `Employs(x, y)` is the most general query, asking for any employment relationship.

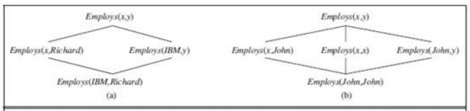


Figure 9.2 (a) The subsumption lattice whose lowest node is *Employs*(*IBM*, *Richard*). (b) The subsumption lattice for the sentence *Employs*(*John*, *John*).

- **Child Nodes:** A child node in the lattice is derived from its parent node by a single substitution. For example, `Employs(x, y)` (parent) can lead to `Employs(x, Richard)` (child) by substituting `y` with `Richard`.

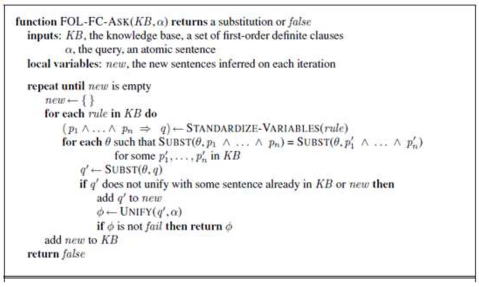
8. Efficiency and Complexity of Indexing

- **Number of Nodes:** The lattice structure can become complex. For a predicate with n arguments, the lattice can have $O(2^n)$ nodes. If the terms involve function symbols, the lattice can grow exponentially with the size of the terms.
- **Cost of Indexing:** Maintaining such a large number of indices can become inefficient. At some point, the overhead of maintaining indices outweighs the benefits of fast retrieval.
- **Fixed vs. Adaptive Policy:** To manage this complexity, systems can use a fixed policy (e.g., indexing only on the predicate and each argument) or an adaptive policy that creates indices based on the types of queries frequently asked.

Forward chaining

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 in the forward direction to extract more data until a goal is reached.

- It is a bottom-up approach for drawing the inferences.
- 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. And is also called as data-driven

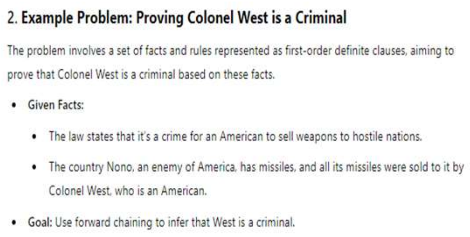


```
function FOL-FC-ASK($KB$, $\alpha$) returns a substitution or $false$
  inputs: $KB$, the knowledge base, a set of first-order definite clauses
          $\alpha$, the query, an atomic sentence
  local variables: $new$, the new sentences inferred on each iteration

  repeat until $new$ is empty
      $new \leftarrow \{\}$
      for each $rule$ in $KB$ do
          $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES($rule$)
          for each $\theta$ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
                      for some $p'_1, \ldots, p'_n$ in $KB$
              $q' \leftarrow$ SUBST($\theta, q$)
              if $q'$ does not unify with some sentence already in $KB$ or $new$ then
                  add $q'$ to $new$
                  $\phi \leftarrow$ UNIFY($q', \alpha$)
                  if $\phi$ is not $fail$ then return $\phi$
      add $new$ to $KB$
  return $false$
```

Forward Chaining Example

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen."

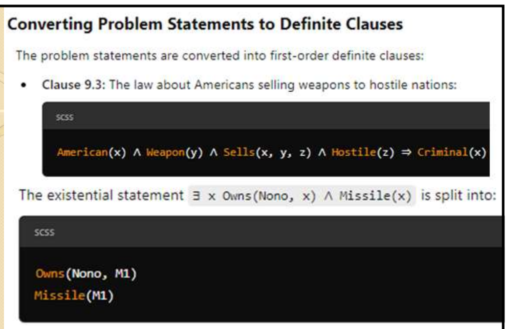
Robert is criminal.



2. Example Problem: Proving Colonel West is a Criminal

The problem involves a set of facts and rules represented as first-order definite clauses, aiming to prove that Colonel West is a criminal based on these facts.

- Given Facts:
  - The law states that it's a crime for an American to sell weapons to hostile nations.
  - The country Nono, an enemy of America, has missiles, and all its missiles were sold to it by Colonel West, who is an American.
- **Goal:** Use forward chaining to infer that West is a criminal.



Converting Problem Statements to Definite Clauses

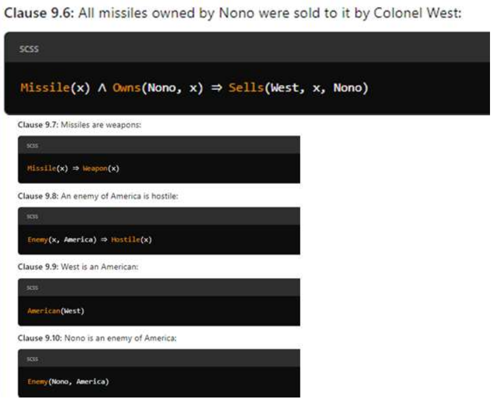
The problem statements are converted into first-order definite clauses:

- **Clause 9.3:** The law about Americans selling weapons to hostile nations:

```scss
American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal(x)
```

The existential statement `∃ x Owns(Nono, x) ∧ Missile(x)` is split into:

```scss
Owns(Nono, M1)
Missile(M1)
```



Clause 9.6: All missiles owned by Nono were sold to it by Colonel West:

```scss
Missile(x) ∧ Owns(Nono, x) ⇒ Sells(West, x, Nono)
```

Clause 9.7: Missiles are weapons:

```scss
Missile(x) ⇒ Weapon(x)
```

Clause 9.8: An enemy of America is hostile:

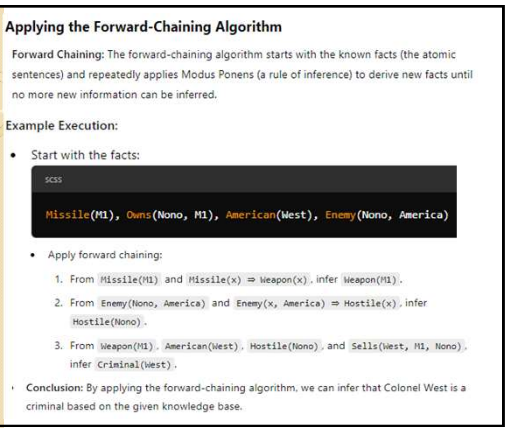
```scss
Enemy(x, America) ⇒ Hostile(x)
```

Clause 9.9: West is an American:

```scss
American(West)
```

Clause 9.10: Nono is an enemy of America:

```scss
Enemy(Nono, America)
```



Applying the Forward-Chaining Algorithm

Forward Chaining: The forward-chaining algorithm starts with the known facts (the atomic sentences) and repeatedly applies Modus Ponens (a rule of inference) to derive new facts until no more new information can be inferred.

Example Execution:

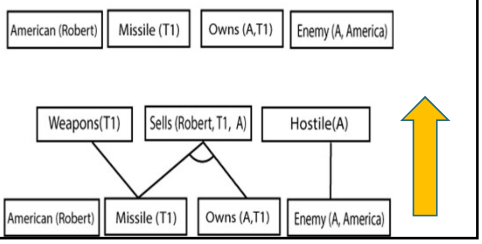
- Start with the facts:

```scss
Missile(M1), Owns(Nono, M1), American(West), Enemy(Nono, America)
```

- Apply forward chaining:
  1. From `Missile(M1)` and `Missile(x) ⇒ Weapon(x)`, infer `Weapon(M1)`.
  2. From `Enemy(Nono, America)` and `Enemy(x, America) ⇒ Hostile(x)`, infer `Hostile(Nono)`.
  3. From `Weapon(M1)`, `American(West)`, `Hostile(Nono)`, and `Sells(West, M1, Nono)`, infer `Criminal(West)`.
- **Conclusion:** By applying the forward-chaining algorithm, we can infer that Colonel West is a criminal based on the given knowledge base.

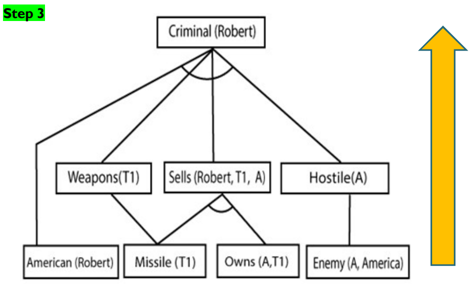
Forward Chaining Proof

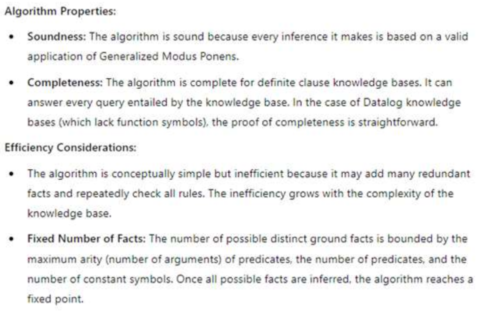
Step I



Continued…

Step 3





Algorithm Properties:

- **Soundness:** The algorithm is sound because every inference it makes is based on a valid application of Generalized Modus Ponens.
- **Completeness:** The algorithm is complete for definite clause knowledge bases. It can answer every query entailed by the knowledge base. In the case of Datalog knowledge bases (which lack function symbols), the proof of completeness is straightforward.

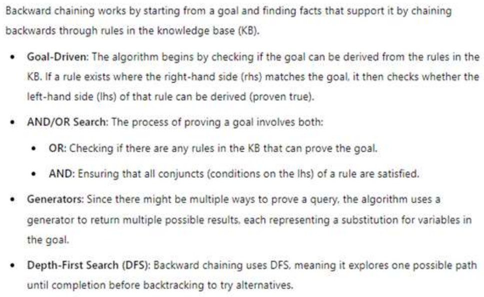
Efficiency Considerations:

- The algorithm is conceptually simple but inefficient because it may add many redundant facts and repeatedly check all rules. The inefficiency grows with the complexity of the knowledge base.
- **Fixed Number of Facts:** The number of possible distinct ground facts is bounded by the maximum arity (number of arguments) of predicates, the number of predicates, and the number of constant symbols. Once all possible facts are inferred, the algorithm reaches a fixed point.

Backward Chaining

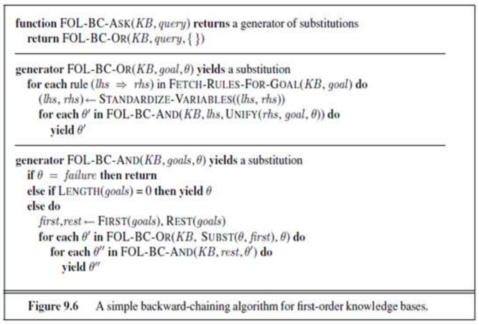
Backward-chaining is also known as a backward deduction or backward reasoning method when using an inference engine. A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

- It is known as a top-down approach.
- 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.



Backward chaining works by starting from a goal and finding facts that support it by chaining backwards through rules in the knowledge base (KB).

- **Goal-Driven:** The algorithm begins by checking if the goal can be derived from the rules in the KB. If a rule exists where the right-hand side (rhs) matches the goal, it then checks whether the left-hand side (lhs) of that rule can be derived (proven true).
- **AND/OR Search:** The process of proving a goal involves both:
  - **OR:** Checking if there are any rules in the KB that can prove the goal.
  - **AND:** Ensuring that all conjuncts (conditions on the lhs) of a rule are satisfied.
- **Generators:** Since there might be multiple ways to prove a query, the algorithm uses a generator to return multiple possible results, each representing a substitution for variables in the goal.
- **Depth-First Search (DFS):** Backward chaining uses DFS, meaning it explores one possible path until completion before backtracking to try alternatives.



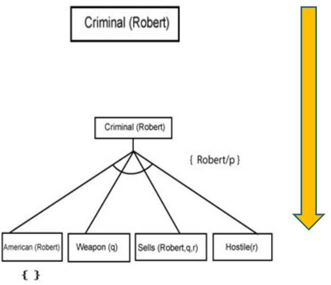
```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ′ in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ′

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first,rest ← FIRST(goals), REST(goals)
    for each θ′ in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ″ in FOL-BC-AND(KB, rest, θ′) do
        yield θ″
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

Backward Chaining

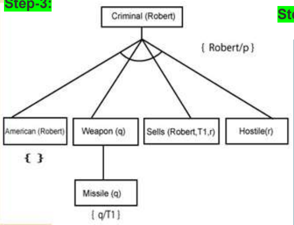
Step-1:

Step-2:

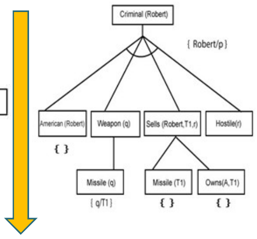


Continued…

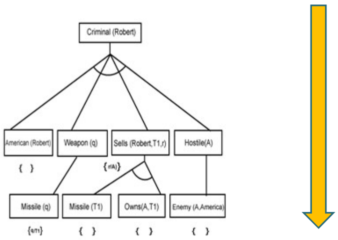
Step-3:



Step-4



Continued…



Resolution

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form**

- **Clause**: Disjunction of literals (an atomic sentence) is called a **clause**. It is also known as a unit clause.
- **Conjunctive Normal Form**: A sentence represented as a conjunction of clauses is said to be **conjunctive normal form** or **CNF**.

Steps for Resolution

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

Example

- Fact
- *All hounds howl at night.*
- *John likes all kind of food.*
- *John likes peanuts.*

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

Step-1: Conversion of facts into first-order logic

- $\forall x\ (HOUND(x) \rightarrow HOWL(x))$
- $\forall x\ \neg\ food(x) \rightarrow likes(John, x)$
- $likes(John, Peanuts)$

a. ∀x: food(x) → likes(John, x)

b. food(Apple) Λ food(vegetables)

c. ∀x ∀y: eats(x, y) Λ ¬ killed(x) → food(y)

d. eats (Anil, Peanuts) Λ alive(Anil).

e. ∀x : eats(Anil, x) → eats(Harry, x)

f. ∀x: ¬ killed(x) → alive(x) } **added predicates.**

g. ∀x: alive(x) →¬ killed(x)

h. likes(John, Peanuts)

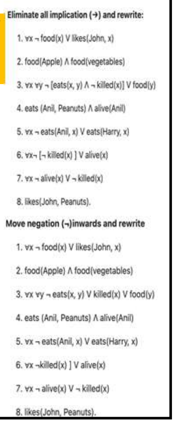
Step-2: Conversion of CNF

Eliminate all implication (→) and rewrite:

- ∀x ¬ HOUND(x) v HOWL(x)
- ∀x ¬ food(x) V likes(John, x)
- likes(John, Peanuts)

Move negation (¬)inwards and rewrite

- ∀x ¬ HOUND(x) v HOWL(x)
- ∀x ¬ food(x) V likes(John, x)
- likes(John, Peanuts)



Eliminate all implication (→) and rewrite:

1. ∀x ¬ food(x) V likes(John, x)
2. food(Apple) Λ food(vegetables)
3. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
4. eats (Anil, Peanuts) Λ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
6. ∀x¬ [¬ killed(x)] V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(John, Peanuts).

Move negation (¬)inwards and rewrite

1. ∀x ¬ food(x) V likes(John, x)
2. food(Apple) Λ food(vegetables)
3. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
4. eats (Anil, Peanuts) Λ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
6. ∀x ¬killed(x)] V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(John, Peanuts).

Step-2 Continued…

Rename variables or standardize variables

- ¬ HOUND(x) v HOWL(x)
- ∀x ¬ food(x) V likes(John, x)
- likes(John, Peanuts)

Eliminate existential instantiation quantifier by elimination (Skolemization)

- ¬ HOUND(x) v HOWL(x)
- ∀x ¬ food(x) V likes(John, x)
- likes(John, Peanuts)

Step-2 Continued…

Drop Universal quantifiers

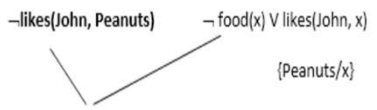
- ¬ HOUND(x) v HOWL(x)
- ¬ food(x) V likes(John, x)
- likes(John, Peanuts)

Step-3: Negate the statement which needs to prove

- In this statement, we will apply negation to the conclusion statements

- ¬ HOUND(x) V HOWL(x)
- ¬ food(x) V likes(John, x)
- ¬ likes(John, Peanuts)

Step-4: Draw Resolution graph



¬likes(John, Peanuts)

¬ food(x) V likes(John, x)

{Peanuts/x}

- First step: **¬likes(John, Peanuts)** , and **likes(John, x)** get resolved(canceled) by substitution of {Peanuts/x}, and we are left with **¬ food(Peanuts)**