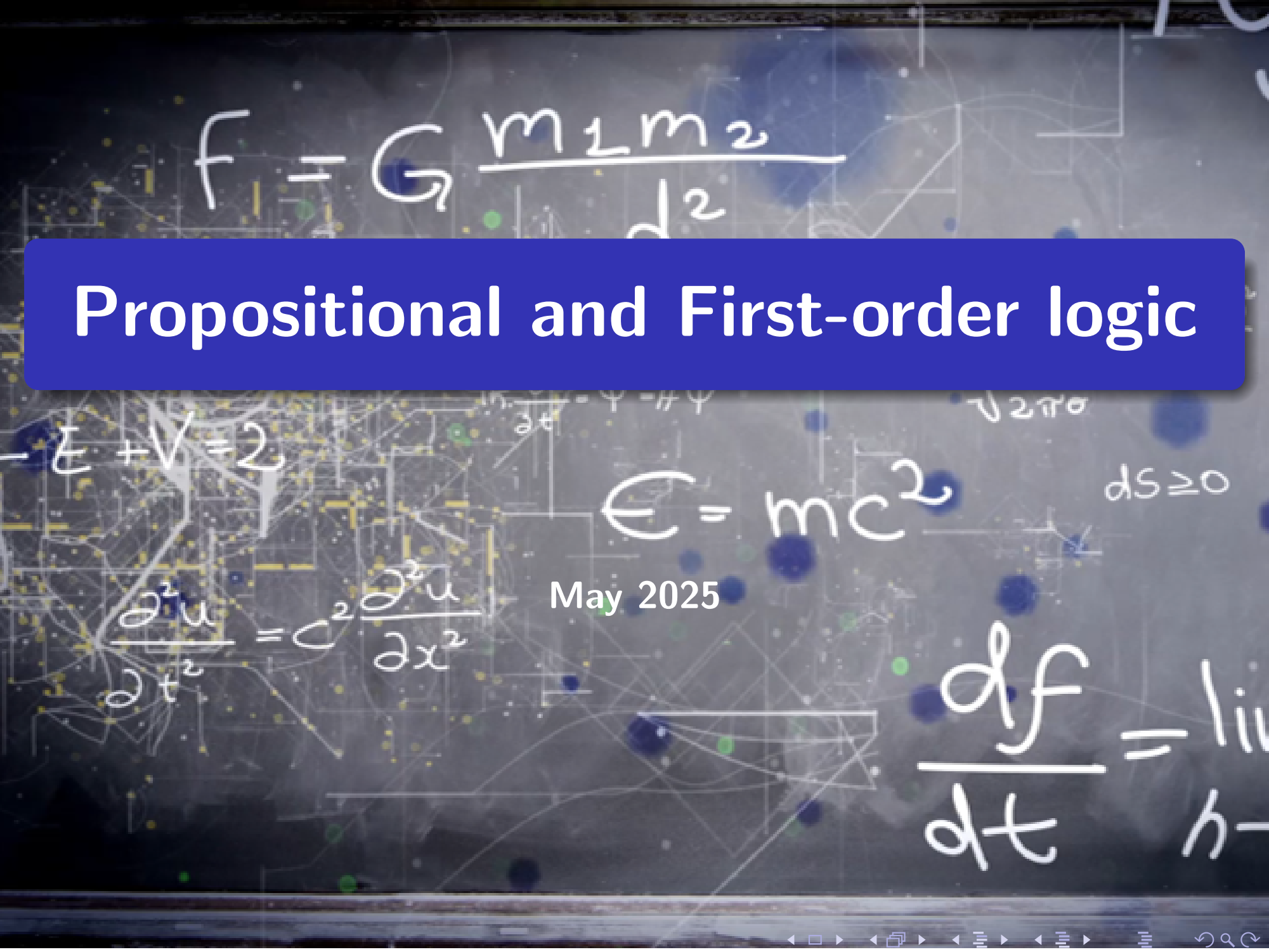


Propositional and First-order logic

May 2025

Topics to be discussed

Part I. Introduction

1. Introduction to Artificial intelligence

Part II. Search and optimisation.

2. Search - basic approaches
3. Search - optimisation
4. Two-player deterministic games
5. Evolutionary and genetic algorithms

Part III. Machine learning and data analysis.

6. Regression, classification and clustering (Part I & II)
8. Artificial neural networks
9. Bayesian models
10. Reinforcement Learning

Part IV. Logic, Inference, Knowledge Representation

11. Propositional logic and predicate logic
12. Knowledge Representation

Part V. AI in Action: Language, Vision

13. AI in Natural language processing
14. AI in Vision and Graphics

Part VI. Summary

15. AI engineering, Explainable AI, Ethics,

Overview

- Introduction to AI and Logic
- Propositional Logic
- First-Order Logic
- PROLOG
- Examples of AI and Logic
- Challenges and Limitations of Logic-Based AI
- Conclusions

Introduction to AI and logic

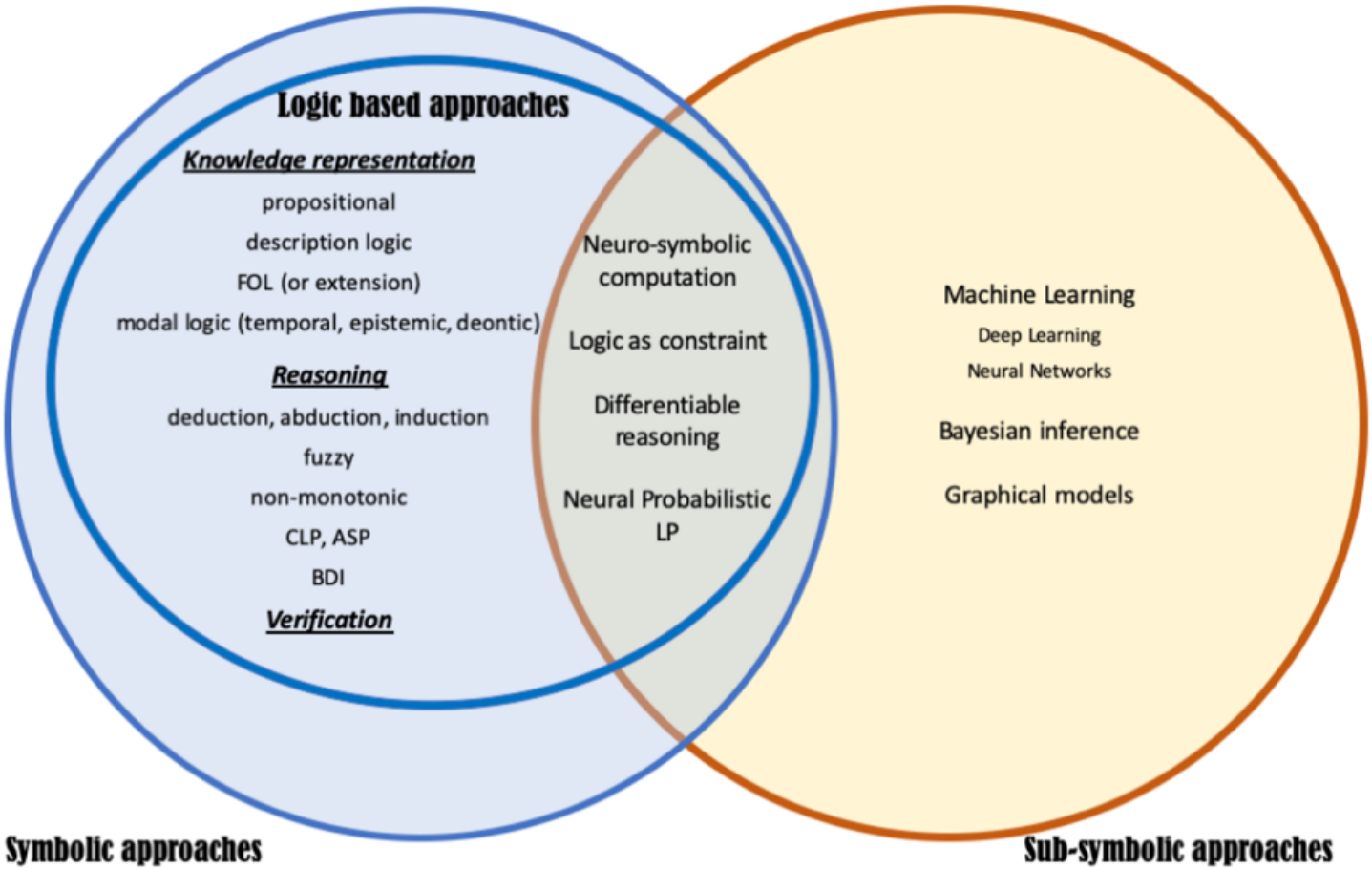
- Logic is a formal system for reasoning about propositions and their relationships.
- In AI, logic has been used to represent knowledge and reason about it.
- Logic-based AI systems use a set of rules and constraints to make inferences and draw conclusions.
- Logic provides a solid foundation for building intelligent systems that can reason and make decisions.
- Some of the popular logic-based AI systems include rule-based systems, constraint satisfaction systems, and automated theorem provers.

History of Logic in AI

Logic has been an important part of AI from the very beginning.

- In the 1950s, researchers like John McCarthy and Marvin Minsky developed the concept of symbolic AI, which used logical reasoning to solve problems.
- In the 1960s, researchers like Alan Newell and Herbert Simon developed the first AI program, the General Problem Solver, which used logical reasoning to solve problems.
- In the 1970s and 1980s, the advent of knowledge representation languages like Prolog and frames allowed for more expressive and complex logical reasoning.
- In 1980s, the development of the situation calculus by Hector Levesque allowed for reasoning about actions and change in AI systems.
- Then, nonmonotonic logics were introduced by researchers like John McCarthy and Raymond Reiter to reason with incomplete or uncertain information.
- Finally, many rule-based systems, which use a set of rules to make decisions, has been implemented and became popular in the 1980s and 1990s.
- The rise of machine learning and statistical methods in the 1990s shifted the focus away from logic-based AI, but logical reasoning remains an important tool in AI today.
- In recent years, there has been a renewed interest in combining logic and machine learning, resulting in the development of systems like neural-symbolic integration and deep logic.

Introduction to AI and Logic (Cont.)



Propositional Logic

Introduction:
Propositional logic, also known as sentential logic or statement logic, is a branch of mathematical logic that deals with propositions and their relationships with one another.

Definition:
Propositional logic is a formal system that uses propositions, variables, and logical connectives to express and manipulate statements. A proposition is a declarative statement that is either true or false, and a variable is a symbol used to represent a proposition.

Logical Connectives

Logical connectives are symbols that connect propositions to form more complex statements. The main logical connectives in propositional logic are:

- **negation** (not): $\neg P$
- **conjunction** (and): $P \wedge Q$
- **disjunction** (or): $P \vee Q$
- **implication** (if...then): $P \rightarrow Q$
- **biconditional** (if and only if): $P \leftrightarrow Q$

where P and Q are propositions.

Truth Tables

Truth tables are used to determine the truth value of a compound proposition based on the truth values of its component propositions. Each row of the truth table represents a possible combination of truth values for the component propositions, and the final column represents the truth value of the compound proposition.

The truth tables for the main logical connectives in propositional logic are:

P	Q	$\neg P$	$P \vee Q$	$P \wedge Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
T	T	F	T	T	T	T
T	F	F	T	F	F	F
F	T	T	T	F	T	F
F	F	T	F	F	T	T

where:

- $\neg P$ represents the negation of P,
- $P \vee Q$ represents the disjunction (or) of P and Q,
- $P \rightarrow Q$ represents the conditional (if-then) statement of P and Q,
- $P \leftrightarrow Q$ represents the biconditional (if-and-only-if) statement of P and Q, and
- $P \wedge Q$ represents the conjunction (and) of P and Q.
- T stands for true and F stands for false.
- the truth values of these propositions are computed based on the truth values of P and Q using the rules of propositional logic.

Syntax and Semantics of Propositional Logic

- **Syntax**: Propositional logic is defined over a set $\mathcal{P}$ of propositional variables, and consists of the following syntactic elements:
  - Propositional variables: $p_1, p_2, \ldots, p_n, \ldots$.
  - Connectives: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), and $\leftrightarrow$ (biconditional).
  - Parentheses: Used to indicate the order of operations.
- **Formulas**: A formula A of propositional logic is defined inductively as follows:
  - If $p_i \in \mathcal{P}$, then p_i is a formula.
  - If A is a formula, then $\neg A$ is a formula.
  - If A and B are formulas, then $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, and $(A \leftrightarrow B)$ are formulas.

Syntax and Semantics of Propositional Logic

- **Semantics**: The semantics of propositional logic is defined over truth assignments to propositional variables, and consists of the following components:
  - Truth values: Each propositional variable p_i can be assigned either the truth value $\top$ (true) or $\bot$ (false).
  - Truth functions: For each connective α, there is a corresponding truth function τ_α that maps truth values to truth values.
- **Truth assignments**: A truth assignment $\mathcal{A}$ is a function that assigns truth values to propositional variables, i.e., $\mathcal{A}(p_i) \in \top, \bot$ for each $p_i \in \mathcal{P}$.
- **Truth values of formulas**: The truth value of a formula A under a truth assignment $\mathcal{A}$, denoted $\mathcal{A}(A)$, is defined inductively as follows:
  - If A is a propositional variable p_i, then $\mathcal{A}(A) = \mathcal{A}(p_i)$.
  - If $A = \neg B$, then $\mathcal{A}(A) = \tau_\neg(\mathcal{A}(B))$.
  - If $A = (B \circ C)$, where $\circ \in \wedge, \vee, \rightarrow, \leftrightarrow$, then $\mathcal{A}(A) = \tau_\circ(\mathcal{A}(B), \mathcal{A}(C))$.

Logical Properties of Propositional Logic

- **Logical entailment**: A formula A logically entails a formula B if and only if for every truth assignment $\mathcal{A}$, if $\mathcal{A}(A) = \top$, then $\mathcal{A}(B) = \top$. In this case, we write $A \models B$.
- **Logical equivalence**: Two formulas A and B are logically equivalent if and only if for every truth assignment $\mathcal{A}$, $\mathcal{A}(A) = \mathcal{A}(B)$. In this case, we write $A \equiv B$.
- **Satisfiability**: A formula A is satisfiable if and only if there exists a truth assignment $\mathcal{A}$ such that $\mathcal{A}(A) = \top$. Otherwise, A is unsatisfiable.

Entailment, Contradiction, and Contingency in Propositional Logic

- Given two formulas P and Q in propositional logic:
  - P **entails** Q if and only if every truth assignment that satisfies P also satisfies Q. We write this as $P \models Q$.
  - P and Q are **equivalent** if and only if they have the same truth value for every truth assignment. We write this as $P \equiv Q$.
  - P and Q are **contradictory** if and only if there is no truth assignment that satisfies both P and Q. We write this as $P \wedge Q \models \bot$, where $\bot$ represents false.
  - P and Q are **contingent** if and only if there is at least one truth assignment that satisfies P and at least one truth assignment that satisfies $\neg P$. Equivalently, there is at least one truth assignment that satisfies Q and at least one truth assignment that satisfies $\neg Q$.
- Examples:
  - $P \wedge Q \models P$
  - $P \vee Q \equiv Q \vee P$
  - $P \wedge \neg P \models \bot$
  - $P \vee Q$ is **contingent**, since it is true for some truth assignments and false for others.

Inference Rules in Propositional Logic

Inference rules are used to derive new propositions from existing propositions. Some common inference rules in propositional logic include:

- Modus ponens: $\frac{P \to Q, P}{Q}$
- Modus tollens: $\frac{P \to Q, \neg Q}{\neg P}$
- Hypothetical syllogism: $\frac{P \to Q, Q \to R}{P \to R}$
- Disjunctive syllogism: $\frac{P \vee Q, \neg P}{Q}$ or $\frac{P \vee Q, \neg Q}{P}$
- Addition: $\frac{P}{P \vee Q}$ or $\frac{Q}{P \vee Q}$
- Simplification: $\frac{P \wedge Q}{P}$ or $\frac{P \wedge Q}{Q}$
- Conjunction: $\frac{P, Q}{P \wedge Q}$
- Constructive dilemma: $\frac{P \to Q, R \to S, P \vee R}{Q \vee S}$

where P, Q, and R are propositions.

Formulas and Clauses in Propositional Logic

- In propositional logic, a **formula** is a statement that can be either true or false.
- A formula can be constructed using variables, logical operators, and parentheses. For example, $(P \wedge Q) \rightarrow R$ is a formula.
- A **clause** is a disjunction of literals, where a literal is either a variable or its negation. For example, $(\neg P \vee Q \vee R)$ is a clause.
- A formula can be converted to a set of clauses using a process called **clause normal form (CNF) conversion**.
- The CNF convertion process involves the elimination of logical operators other than conjunction (and), disjunction (or), and negation (not) by applying the distributive law and De Morgan's laws.

Converting Formula to CNF

Algorithm Convertion into CNF in propositional logic

Input: A formula F in propositional logic
Output: An equivalent formula F' in CNF

Step 1: Eliminate implications:
Replace $p \to q$ with $\neg p \vee q$
Step 2: Move negations inwards:
Apply De Morgan's laws and double negation elimination
Step 3: Distribute disjunctions over conjunctions:
Apply the distributive law repeatedly until no more distributive steps can be taken
Step 4: Simplify:
Remove tautologies and duplicate clauses
Output: F' is the resulting formula

Horn Clauses in Propositional Logic

- A Horn clause is a logical formula of the form:

$$(A_1 \wedge A_2 \wedge ... \wedge A_n) \rightarrow B$$

  where $A_1, A_2, \ldots, A_n, B$ are propositional atoms.
- A Horn clause is said to be **definite** if it contains at most one positive literal (i.e., B is a positive literal).
- A Horn clause is said to be **goal-reducing** if its conclusion B is a new goal that can be added to the set of goals.

Examples: (which one is a Horn clause?)

- $(P \wedge Q) \rightarrow R$
- $P \rightarrow (Q \rightarrow R)$
- P
- $\neg Q \rightarrow \neg P$
- $P \rightarrow Q$
- $\neg Q$

Inference Algorithm for Horn Clauses

- To perform inference on a knowledge base consisting of Horn clauses, we can use the Modus Ponens inference rule.
- Here is an algorithm that uses the Modus Ponens inference rule to infer new propositions from a knowledge base consisting of Horn clauses:

Algorithm: Inference on Horn Clauses

1. Initialize a set of propositions P with the propositions in the knowledge base.
2. Repeat the following until no new propositions can be inferred:
   - For each Horn clause $A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow B$ in the knowledge base:
     - If $A_1, A_2, \ldots, A_n \subseteq P$, then infer B and add it to P.
3. Return the set of inferred propositions P.

- This algorithm repeatedly applies the Modus Ponens inference rule to each Horn clause in the knowledge base until no new propositions can be inferred.
- The resulting set of inferred propositions is guaranteed to be sound and complete with respect to the original knowledge base, which means that it always produces true conclusions if the premises are true (**soundness**), and it is capable of deriving all true conclusions that can be inferred from the premises (**completeness**).

Backward and Forward Chaining using Resolution

Resolution in propositional logic facilitates two main inferencing approaches: **backward chaining** and **forward chaining**.

Forward Chaining (Data-Driven):

1. **Start:** From known facts or premises.
2. **Resolution:** Apply resolution to derive new conclusions from existing clauses.
3. **Repeat:** Continue until a goal or conclusion is derived or all possibilities are exhausted.
4. **Example:** Given A, $A \rightarrow B$, $B \rightarrow C$; derive C.

Backward Chaining (Goal-Driven):

1. **Start:** From the goal, assume its negation.
2. **Resolution:** Use resolution to find contradictions to this assumption.
3. **Trace Back:** Link back to initial facts to support the goal.
4. **Contradiction:** Derive the empty clause, proving the goal by contradiction.
5. **Example:** To prove C, start with $\neg C$, resolve to show $\neg B$ and $\neg A$ follow; check against facts.

Comparison:

- **Forward Chaining:** Efficient when the path to conclusions is clear.
- **Backward Chaining:** Useful when the goal is clear but the path is not.
- **Resolution** in both cases is a sound and complete inference rule for propositional logic.

Inference by Resolution (Forward Chaining)

Algorithm Resolution Inference Algorithm

Input: A set of propositional clauses S and a propositional formula α
Output: True if α is entailed by S, False otherwise

```
while α is not in S do
    foreach pair of clauses C_i, C_j ∈ S do
        resolvents ← the set of all clauses obtained by applying the resolution rule to C_i and C_j
        foreach clause C_k ∈ resolvents do
            if C_k is not a tautology and C_k is not in S then
                Add C_k to S
            end
        end
    end
end
if α is in S then
    return True
end
else
    return False
end
```

Soundness and Completeness in Propositional Logic

- A deductive system for propositional logic is **sound** if and only if every provable formula is logically valid. In other words, if a formula P can be derived from a set of premises Γ using the deductive system, then P is logically valid given Γ. We write this as $\Gamma \vdash P \Rightarrow \Gamma \models P$.

- A deductive system for propositional logic is **complete** if and only if every logically valid formula is provable. In other words, if a formula P is logically valid given Γ, then P can be derived from Γ using the deductive system. We write this as $\Gamma \models P \Rightarrow \Gamma \vdash P$.

First-Order Logic: Introduction

- First-order logic (FOL) is a formal system used to represent and reason about relationships between objects, and is commonly used in mathematics, computer science, and philosophy.
- In FOL, we use predicates to represent properties of objects, and quantifiers to make statements about groups of objects.
- Predicates are functions that map objects to truth values, and are denoted by capital letters such as P or Q. For example, $P(x)$ might represent the property x is a prime number."
- Quantifiers are used to make statements about all or some of the objects in a domain. The universal quantifier ($\forall$) is used to make statements about all objects, while the existential quantifier ($\exists$) is used to make statements about at least one object.
- For example, the statement $\forall x\, P(x)$ means "All objects have the property P", while the statement $\exists x, P(x)$ means "There exists an object with the property P."

First-order Logic: Syntax and Semantics

- Syntax:
  - Predicates: $P, Q, R, \ldots$
  - Variables: $x, y, z, \ldots$
  - Constants: $a, b, c, \ldots$
  - Functions: $f, g, h, \ldots$
  - Connectives: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence),
  - Quantifiers: $\forall$ (universal quantification), $\exists$ (existential quantification)
  - Parentheses for grouping
- Semantics:
  - Interpretations:
    - domain of discourse,
    - assignment of objects to constants,
    - assignment of functions to operations,
    - assignment of truth values to predicates
  - Truth values: $\top$ (true), $\bot$ (false)

Terms and Atoms

- Terms: Variables, constants, and functions applied to terms.
- Atoms: Predicates applied to terms.

Examples:

- Term examples:
  - Variable: x, y, z
  - Constant: a, b, c
  - Function application: $f(x)$, $g(y, z)$, $h(a, b)$
  - Complex term: $f(g(a, b), c)$
- Atom examples:
  - Predicate application: $P(x)$, $Q(x, y)$, $R(f(x), a)$
  - Complex atom: $P(f(x), g(y, z))$
  - Multiple predicates and terms: $P(x) \land Q(y)$, $R(f(x), g(y)) \lor S(a)$

Literals and Clauses

- Literals: Atoms and negated atoms.
  - Atoms: Predicates applied to terms, representing basic assertions.
  - Negated atoms: Negations of atoms, indicating the negation of a basic assertion.
  - Examples of literals:
    - Atoms: $P(x)$, $Q(y)$, $R(z)$
    - Negated atoms: $\neg P(x)$, $\neg Q(y)$, $\neg R(z)$
- Clause: A disjunction of literals, representing a logical statement or rule.
  - Examples of clauses:
    - $P(x) \vee Q(y)$
    - $\neg Q(y) \vee R(z)$
    - $P(x) \vee \neg R(z) \vee \neg Q(y)$
- Horn clause: A clause that contains at most one positive literal.
  - Examples of Horn clauses:
    - $P(x) \rightarrow Q(y)$
    - $P(x) \vee \neg Q(y) \rightarrow R(z)$

Substitution and Unification

- **Substitution**: Substitution involves replacing variables with terms.
  - Example 1: $\sigma = [x \rightarrow f(y)]$ is a substitution. It assigns the term $f(y)$ to the variable x.
  - Example 2: $\sigma(P(x), Q(y)) = P(f(y)), Q(y)$. The substitution σ replaces x with $f(y)$ in $P(x)$ and leaves $Q(y)$ unchanged.

- **Unification**: Unification is the process of finding a most general unifier (MGU) for two terms, making them identical by applying substitutions.
  - Example 3: $\mathit{unify}(f(x), f(a),) = x \rightarrow a$. The terms $f(x)$ and $f(a)$ can be unified by substituting x with a.
  - Example 4: $\mathit{unify}(f(x), g(a),) = \perp$ (no MGU). The terms $f(x)$ and $g(a)$ cannot be unified, as they have different function symbols and cannot be made identical.
  - Example 5: $\mathit{unify}(f(g(x)), f(g(a)),) = x \rightarrow a$. The terms $f(g(x))$ and $f(g(a))$ can be unified by replacing the variable x with the constant a.
  - Example 6: $\mathit{unify}(f(x, y), f(g(a), z),) = x \rightarrow g(a), y \rightarrow z$. The terms $f(x, y)$ and $f(g(a), z)$ can be unified by replacing x with $g(a)$ and y with z.
  - Example 7: $\mathit{unify}(f(g(x), h(y)), f(g(a), h(b)),) = x \rightarrow a, y \rightarrow b$. The terms $f(g(x), h(y))$ and $f(g(a), h(b))$ can be unified by replacing x with a and y with b.

Skolemization in First-Order Logic

- **Skolemization** is a process in FOL that eliminates existential quantifiers by introducing new skolem constants or skolem functions.
- **Skolem constants** are individual constants that represent specific objects. For example, if we have an existential quantifier $\exists x P(x)$, we can replace it with a skolem constant c to obtain $P(c)$. The skolem constant c represents a specific object that satisfies the predicate P.
- **Skolem functions** are functions that generate new objects. For example, if we have an existential quantifier $\forall y \exists x P(x)$, we can introduce a skolem function f to obtain $P(f(y))$, where y is a variable that binds the outermost quantifier that includes the existential quantifier. The skolem function f generates a new object that satisfies the predicate P.
- **Example 1: Skolem Constants**
  Consider the FOL sentence: $\exists y R(x, y)$. To skolemize, we introduce a skolem constant c and replace the existential quantifier with it, obtaining $R(x, c)$.
- **Example 2: Skolem Functions**
  Consider the FOL sentence: $\forall x \exists y R(x, y)$. To skolemize, we introduce a skolem function f and replace the existential quantifier with it, obtaining $\forall x R(x, f(x))$.
- **Skolemization preserves the satisfiability and logical consequences** of a formula, meaning that the skolemized formula is satisfiable if and only if the original formula is satisfiable, and the skolemized formula logically implies the original formula, and vice versa.

Conversion of a first-order logic formula to CNF

- Converting to **conjunctive normal form (CNF)** is a common step in many automated theorem proving systems.
- CNF is a conjunction of clauses, where each clause is a disjunction of literals (i.e., atomic formulas or their negations).
- **Example:**
  - A formula $P(x) \rightarrow \exists y.Q(y) \vee R(x, y)$ can be converted to CNF as follows:
    - Rewrite the implication as a disjunction: $\neg P(x) \vee \exists y.Q(y) \vee R(x, y)$.
    - Move the quantifier out using Skolemization: $\neg P(x) \vee Q(f(x)) \vee R(x, f(x))$

Prenex Normal Form (PNF)

Definition:

Prenex Normal Form (PNF) is a form of presenting logical formulas where all the quantifiers are moved to the front, followed by a quantifier-free matrix.

Characteristics:

- All quantifiers ($\forall$ and $\exists$) appear at the beginning of the formula.
- The core of the formula containing logical connectives ($\wedge$, $\vee$, etc.) without any quantifiers.

Conversion Steps:

1. Eliminate implications: $A \rightarrow B$ is replaced by $\neg A \vee B$.
2. Move negations inward using De Morgan's laws and negation rules.
3. Standardize variables to avoid overlap in variable scoping.
4. Rearrange quantifiers, moving them to the front while respecting logical dependencies.

Example Conversion:

$$\text{Original: } (A(x) \rightarrow \forall y B(y)) \wedge \exists z C(z, x)$$

$$\text{Step 1: } (\neg A(x) \vee \forall y B(y)) \wedge \exists z C(z, x)$$

$$\text{Final: } \forall y \exists z (\neg A(x) \vee B(y)) \wedge C(z, x)$$

Conversion of a First-Order Logic Formula to CNF

Algorithm Conversion of a First-Order Logic Formula to CNF

Input: A first-order logic formula F
Output: A CNF version of F

Convert F to prenex normal form
  Apply Skolemization to F to eliminate existential quantifiers
  Eliminate universal quantifiers (as they are implicit in CNF)
  Use logical equivalences to eliminate implications and biconditionals from F
  for *each operation necessary to apply De Morgan's laws and double negation elimination* **do**
  | Apply the operation to F
  |
end
for *each disjunction and conjunction in* F **do**
  | Distribute $\wedge$ over $\vee$ to transform F into the final CNF form
  |
end
return *The set of clauses in CNF*

Resolution and Refutation

- The idea of **resolution** is to derive new clauses by resolving (i.e., unifying and simplifying) existing clauses that contain complementary literals.
- Example:
  - Having the following set of formulas in the CNF representation: $(\neg P(x) \vee Q(f(x)))$, $(P(x) \vee R(x, f(x)))$.
  - We can apply resolution to this set of clauses, and derive a new clause by unifying and simplifying the two clauses: $Q(f(x)) \vee R(x, f(x))$.
- **Refutation** is the process of trying to prove a statement by deriving a contradiction from its negation.
- Example:
  - Given the following set of clauses: $P(x) \vee Q(y), \neg Q(f(a)) \vee R(z), \neg R(z)$.
  - To prove $P(x)$ by contradiction, we add $\neg P(x)$ as a new clause and derive a contradiction by applying resolution to the set of clauses.
  - The new set of clauses becomes: $P(x) \vee Q(y), \neg Q(f(a)) \vee R(z), \neg R(z), \neg P(x)$.
  - Resolving on $\neg Q(f(a)) \vee R(z)$ and $P(x) \vee Q(y)$, we obtain the new clause: $P(x) \vee R(z)$.
  - Resolving on $\neg R(z), \neg P(x)$ and $P(x) \vee R(z)$, we obtain the empty clause $\square$, which means that the set of clauses is contradictory and $\neg P(x)$ is refuted. Therefore, $P(x)$ is proven.

Resolution inference in First Order Logic (by Refutation)

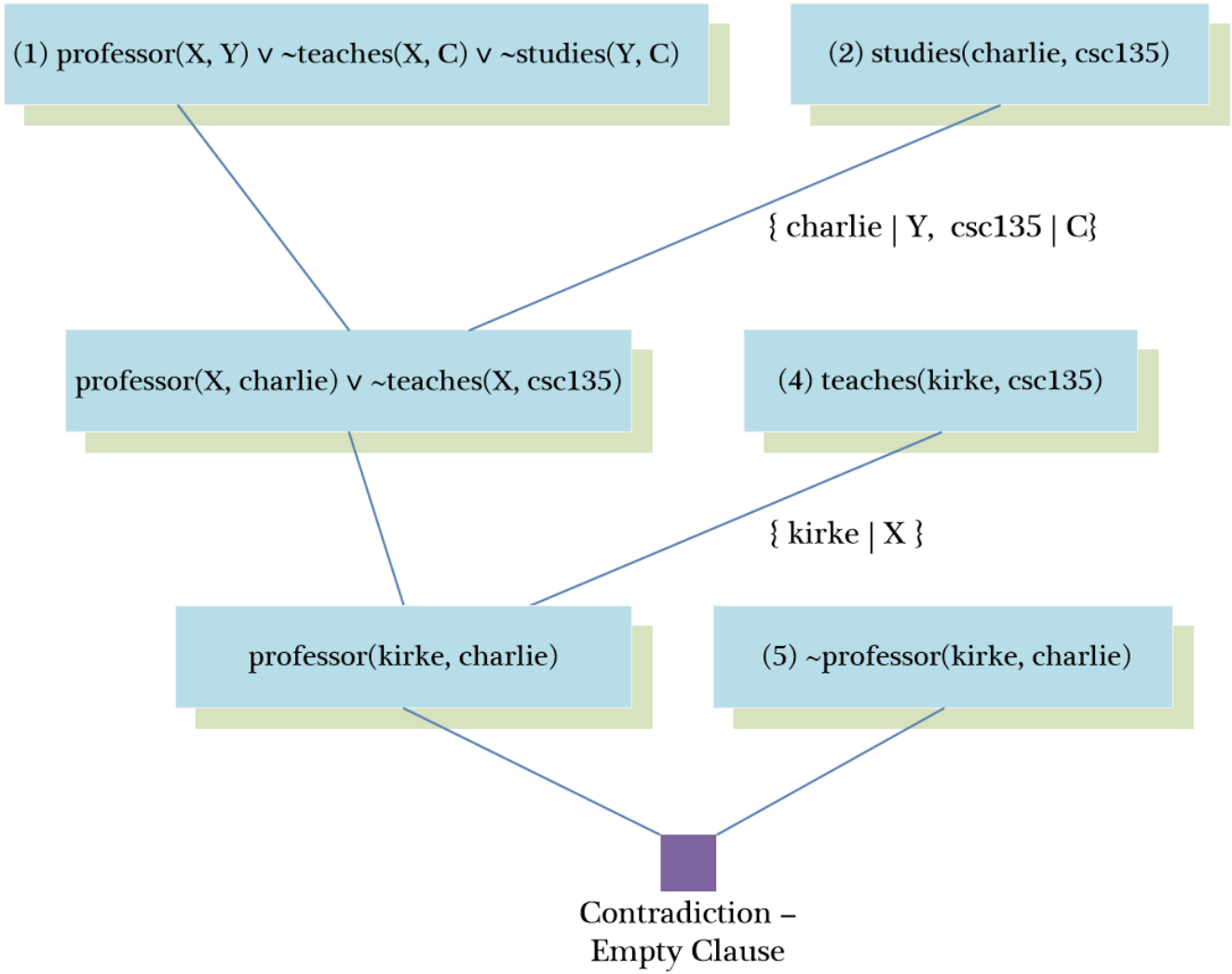
Algorithm Resolution inference rule for FOL

Input: Knowledge base KB and query Q
Output: True if Q is entailed by KB, False otherwise

Function `Refutation(`*KB*, *Q*`)`:

- **for** *each sentence* α *in KB* **do**
  - Convert α to prenex normal form
  - Skolemize α
  - Convert α to conjunctive normal form (CNF)
- **end**
- Convert $\neg Q$ to prenex normal form
- Skolemize $\neg Q$
- Convert $\neg Q$ to conjunctive normal form
- $\mathcal{C} \leftarrow$ Conjuncts of KB and $\neg Q$
- **while** *there exists a pair of clauses* C_i *and* C_j *that can be resolved* **do**
  - Resolve C_i and C_j using unification
    - **if** *resolvent is empty* **then**
      - **return** True
    - **end**
  - Add resolvent to $\mathcal{C}$
- **end**
- **return** False

Resolution inference by contradiction



https://athena.ecs.csus.edu/ mei/logicp/unification-resolution/trees-of-resolution-examples.html

Horn Clauses in First-Order Logic

- A Horn clause in first-order logic is a logical formula of the form:

$$\forall x_1 \ldots \forall x_m((A_1 \wedge \ldots \wedge A_n) \rightarrow B)$$

  where $A_1, \ldots, A_n$ and B are atomic formulas in first-order logic, and at most one of them is positive.

- Horn clauses are important in AI because they can be used to represent implications and logical rules in a concise and efficient manner. In particular, they are useful for tasks such as expert systems, diagnosis, planning, and natural language processing.

Examples of Horn clauses:

- $\forall x(P(x) \rightarrow Q(x))$
- $\forall x \forall y(P(x, y) \wedge Q(x, y) \rightarrow R(x))$

Forward Inference with Horn Clauses and Modus Ponens

Algorithm Forward Inference in FOL with Horn Clauses and Modus Ponens

Data: A knowledge base KB of Horn clauses in FOL
Result: A set of inferred facts
Apply Prenex normal form to all clauses
Apply Skolemization to all existential quantifiers using unification and substitution
Convert all clauses to Horn definite form
while *New facts are inferred* **do**
    Initialize an empty set to hold inferred facts
    for *Each Horn clause in the knowledge base KB* **do**
        Let $A_1, A_2, \ldots, A_n$ be the antecedents of the Horn clause, and let B be the consequent
        if $A_1, A_2, \ldots, A_n$ *are known to be true* **then**
            Infer B
            Add B to the set of inferred facts
    Add the inferred facts to the knowledge base KB;

This algorithm assumes that KB is consistent, meaning that there are no contradictions in KB. If KB is inconsistent, the algorithm will not terminate.

An Introduction to PROLOG Programming Language

- PROLOG stands for "PROgramming in LOGic"
- As a logic programming language it has been widely used in AI and natural language processing
- Based on first-order logic and the notion of logical inference
- Useful for representing knowledge and solving problems in domains such as expert systems, natural language processing, and robotics
- Developed in the 1970s by Alain Colmerauer and his colleagues at the University of Aix-Marseille in France
- Inspired by the concept of natural language understanding, PROLOG aimed to create a programming language that could mimic the way humans reason
- PROLOG's syntax is simple and declarative, making it easy to write and understand logic-based programs

Basic Syntax and Semantics of PROLOG

- The basic building blocks of PROLOG are terms
  - Terms can be atoms, variables, or complex structures
  - Atoms are simple, constant values (e.g., ‘hello‘, ‘world‘)
  - Variables are represented by a sequence of letters or digits starting with a capital letter (e.g., ‘X‘, ‘Y‘, ‘Z‘)
  - Complex structures are composed of a functor and one or more arguments (e.g., ‘parent(john,mary)‘)
- PROLOG programs are made up of a set of clauses
  - A clause is either a fact or a rule
  - Facts are statements of the form ‘p(a, b, c).‘, which assert that the predicate ‘p‘ is true of the arguments ‘a‘, ‘b‘, and ‘c‘
  - Rules are statements of the form ‘p(X, Y) :- q(X), r(Y).‘, which assert that the predicate ‘p‘ is true of the arguments ‘X‘ and ‘Y‘ if the predicates ‘q‘ and ‘r‘ are true of ‘X‘ and ‘Y‘, respectively
- In PROLOG, we can ask queries to the program using the ‘?-‘ operator
  - For example, ‘?- parent(john, mary).‘ would ask whether ‘john‘ is the parent of ‘mary‘
  - PROLOG uses logical inference to find solutions to queries based on the facts and rules in the program

Example PROLOG Program

Knowledgebase

```
teaches(joe, calculus).
teaches(ed, programming).
teaches(ed, cs).
enrolled(jim, programming).
enrolled(lisa, cs).
enrolled(Student, Course) :- enrolled(OtherStudent, Course), teaches(Teacher, Course).
course(cs, Course).
course(history, ancient_civilizations).
course(physics, mechanics).
course(math, calculus).
```

Example queries and answers

```
?- enrolled(jim, calculus).
No
?- teaches(Teacher, programming).
Teacher = ed
?- enrolled(lisa, Course).
Course = cs
?- enrolled(Student, cs).
Student = lisa
?- enrolled(lisa, Course), teaches(joe, Course).
No
```

Logical Inference and Reasoning in PROLOG

- PROLOG uses logical inference to find solutions to queries
  - The inference engine searches through the database of facts and rules to find matches for the query
  - It uses unification to match the variables in the query with the corresponding terms in the database
  - If it finds a match, it returns the substitution that makes the query true
  - If it doesn't find a match, it fails and backtracks to try another solution
- PROLOG supports both forward and backward reasoning
  - Forward reasoning starts with the facts and applies rules to derive new conclusions
  - Backward reasoning starts with the query and applies rules to derive the facts needed to prove the query
  - Backward reasoning is often used in expert systems and problem-solving applications
- PROLOG also supports meta-programming
  - Meta-programming allows programs to manipulate themselves or other programs at runtime
  - It can be used to implement advanced features such as reflection, introspection, and code generation

Types of Resolution in Prolog

In Prolog, there are two main types of resolution:

1. **SLD resolution** (Selective Linear Definite resolution):
   - Used to evaluate queries against a Prolog program.
   - Uses a top-down approach.
   - Considers only definite clauses (i.e., Horn clauses with at most one positive literal).
   - Generates a resolution tree to explore all possible paths of the computation.
2. **SLG resolution** (Semi-linear resolution for General logic programs):
   - Used to handle non-definite clauses (i.e., clauses with more than one positive literal).
   - Uses a bottom-up approach.
   - Uses a technique called tabling to memoize subgoals and avoid redundant computations.
   - More powerful than SLD resolution and can handle a wider range of programs, including non-stratified negation.

SLD Resolution

SLD resolution (SLD for "Selective Linear Definite") is a form of resolution used in Prolog that systematically searches through the clauses of a program in order to find solutions to a query.

- **Goal**: A query to be solved, written in the form of a predicate.
- **Clause**: A rule or fact that is used to define the behavior of a predicate.
- **Substitution**: A mapping between variables and terms that is used to instantiate predicates and clauses.
- **Backtracking**: The process of undoing a substitution and trying the next possible solution to a goal.

SLD resolution works by starting with the first clause of a program and attempting to unify the query with the head of the clause. If unification succeeds, the body of the clause is added to the list of goals to be solved. The process continues recursively until all goals have been solved or until a goal fails to unify with any remaining clauses.

SLD resolution - pseudocode

Algorithm SLD Resolution for Logic Programming

Input: A logic program P and a query Q

Output: A proof tree for Q in P or failure

Function `SLD_Resolution`(P, Q):

```
if Q is ground and Q ∈ P then
    return success
end
for each ground instance Q' of Q do
    Choose a clause C in P whose head unifies with Q'
    Apply the most general unifier θ to the body of C
    if SLD_Resolution(P, θ(Q')) succeeds then
        return success
    end
end
return failure
```

Summary - Challenges and Limitations of Logic-Based AI

- **Expressivity**: Logic-based AI has limited expressivity compared to other approaches, such as neural networks or deep learning.
- **Scalability**: It can struggle to scale to larger and more complex problems due to its reliance on formal logic and inference rules.
- **Knowledge Acquisition**: It often requires explicit knowledge representation, which can be difficult and time-consuming to acquire and maintain.
- **Uncertainty**: It struggles with representing and reasoning about uncertainty, which is a common feature of real-world problems.
- **Lack of Context**: It may struggle with reasoning about context, as it typically relies on formal rules and does not have the ability to reason about the broader context of a problem.

Summary - Propositional and First-order logic

- Introduction to AI and Logic
- Propositional Logic
- First-Order Logic
- PROLOG
- Examples of AI and Logic
- Challenges and Limitations of Logic-Based AI
- Conclusion and Future Directions

References

1. S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach", Financial Times Prentice Hall, 2019.
2. R. Kowalski, "Logic for problem solving", Books on Demand, 2014
3. M. Flasiński, Introduction to Artificial Intelligence", Springer Verlang, 2016
4. SWI-Prolog - free implementation of the PROLOG, wiki: https://en.wikipedia.org/wiki/SWI-Prolog
5. SWI-Prolog community: https://www.swi-prolog.org/community.html