AI Final Exam Study

Section 1: Intelligent Agents & Foundations

1.1 Definitions of Artificial Intelligence (AI)

- **AI as a Science:** The field that studies the construction and analysis of computational agents that act intelligently.
- **Brain Simulation:** The study of computer systems that attempt to simulate and apply the intelligence of the human brain (e.g., writing a program to recognize objects in an image).
- **Core Capabilities of AI:**
    1. Ability to solve problems.
    2. Ability to act rationally.
    3. Ability to act like humans.

1.2 Conventional Programming vs. AI Programming

- **Conventional Programming:** Relies on a rigid binary formula:

Program = Algorithm+ Data

The program follows strictly predefined steps and cannot handle missing or ambiguous data without manual code modifications.

- **AI Programming:** Relies on knowledge and inference:

Intelligent System = Knowledge Base +Inference Engine

It focuses on heuristics and explanations, enabling decision-making even when data is ambiguous or incomplete.

- **Transforming Conventional to Intelligent:** To convert a conventional program into an intelligent one, you must integrate a **Knowledge Base** (containing domain expertise), an **Inference Engine** (capable of inferring new solutions), and a mechanism to learn from feedback.

1.3 Computational Agents & Environments

- **Agent:** Anything that can be viewed as perceiving its environment through **Sensors** and acting upon that environment through **Actuators**.

- **Agent Limitations:** An agent generally cannot observe the complete state of the world directly, operates with limited memory, and has a constrained amount of time to act.
- **Agent Function:** A mathematical function that maps any given percept sequence to an action:

Agent Function: Percept Sequences to Actions

- **Agent Program:** The actual software execution running on the physical architecture to implement the agent function.

Agent = Architecture+ Program

- **Single-Agent vs. Multi-Agent Environments:**
    - **Single-Agent:** An agent operating alone in an environment where no other agents impact its performance measure (e.g., a robotic vacuum in an empty room).
    - **Multi-Agent:** An environment containing more than one agent where actions interact. Relationships can be **Cooperative** or **Competitive** (e.g., chess players or autonomous cars in heavy traffic).

1.4 Rationality & Autonomy

- **Rationality:** A property of agents that select actions expected to **maximize their performance measure** (expected utility), based on the percept sequence received to date and built-in prior knowledge.
- **Rationality Depends on Four Factors:**
    1. **Performance Measure:** Defines the criterion for success.
    2. **Prior Knowledge:** The agent's embedded understanding of the environment.
    3. **Actions:** The set of permissible actions the agent can perform.
    4. **Percept Sequence:** Everything the agent has perceived up to the current moment.
- **Autonomous Agents:** Agents that rely on and learn from their own experience to compensate for incorrect or incomplete prior knowledge injected by the designer. If an agent relies entirely on the designer's prior knowledge rather than its own percepts, it **lacks autonomy**.

1.5 PEAS Application for Targeted Environments

A) Vacuum-Cleaner World

- **Performance Measure:** Amount of dirt cleaned, time taken, electricity consumed, and noise level.
- **Environment:** Rooms/Locations (e.g., Location A, Location B), cleanliness state (Dirty, Clean).
- **Actuators:** Wheels for locomotion, brushes for sweeping, vacuum suction motor (Actions: Left, Right, Suck, NoOp).
- **Sensors:** Location sensor (identifying the current room), infrared or camera sensors to detect the presence of dirt.

B) Medical Diagnosis System

- **Performance Measure:** Healthy patient recovery, minimized financial/time costs, avoidance of medical malpractice or errors.
- **Environment:** Patient, hospital staff, clinical conditions.
- **Actuators:** Display screen for questions/diagnoses, diagnostic test ordering, prescription writing, and referral systems.
- **Sensors:** Keyboard/touchscreen for inputting symptoms, laboratory test results, and patient responses.

1.6 Properties of Task Environments

- **Fully Observable vs. Partially Observable:** Fully observable if sensors grant access to the complete state of the environment at any given point in time. Partially observable due to noisy, inaccurate, or missing sensor data.
- **Deterministic vs. Stochastic:** Deterministic if the next state of the environment is completely determined by the current state and the action executed by the agent. Otherwise, it is stochastic (uncertain).
- **Episodic vs. Sequential:** In episodic environments, the agent's experience is divided into atomic, independent episodes; current actions do not affect future episodes. In sequential environments, current decisions heavily influence future states.
- **Static vs. Dynamic:** Static if the environment does not change while the agent is deliberating. Dynamic if the environment keeps changing during deliberation (e.g.,

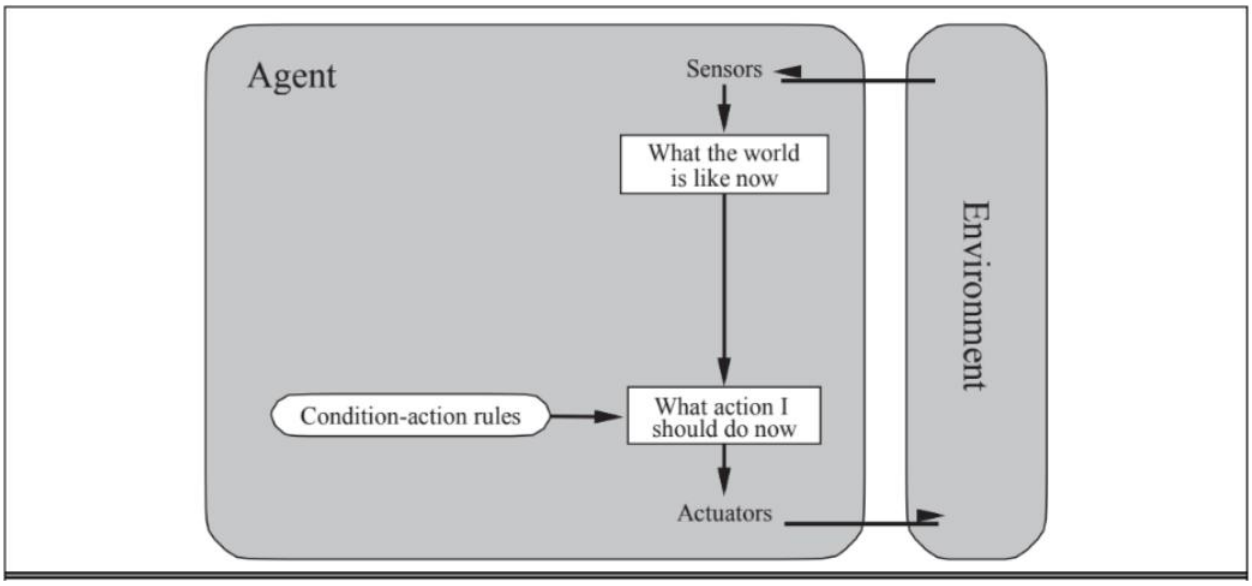
driving a taxi). *Semi-dynamic* environments do not change with time, but the agent's performance score drops the longer it takes to deliberate.

- **Discrete vs. Continuous:** Applies to states, time, percepts, or actions. Chess is discrete (distinct states and turns); driving an automobile is continuous.

1.7 The Five Core Agent Architectures

1. Simple Reflex Agent

- **Description:** Selects actions based *only* on the current percept, completely ignoring the history of past percepts. Driven by strictly defined **Condition-Action Rules** ("If X, then do Y"). It operates successfully only if the environment is **Fully Observable**.
- **Example:** A basic smart vacuum: IF location is dirty -> THEN Suck.
- **Architecture Diagram:**



Pseudocode Structure:

Python

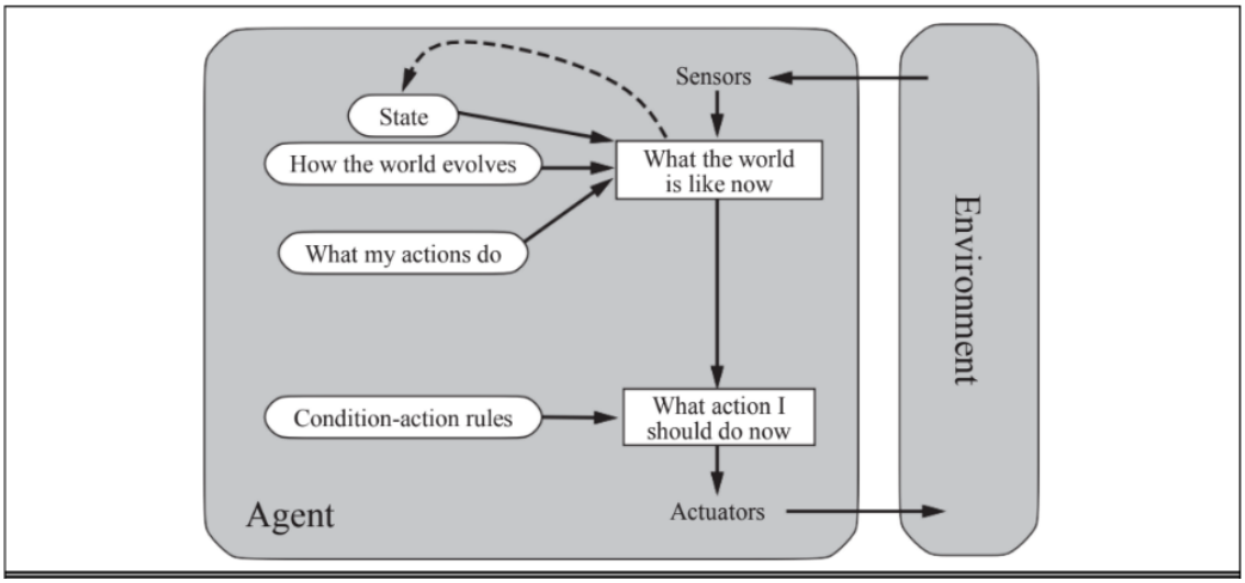
```
def SIMPLE_REFLEX_AGENT(percept):
    # persistent: rules (a set of condition-action rules)
    state = INTERPRET_INPUT(percept)
    rule = RULE_MATCH(state, rules)
```

action = rule.ACTION

return action

2. Model-Based Reflex Agent

- **Description:** Handles **Partially Observable** environments by maintaining an **Internal State** that tracks unobserved aspects of the current world. It incorporates a model detailing how the world evolves independently and how the agent's actions impact the world.
- **Example:** An autonomous car tracking the vehicle ahead; if thick fog blocks the sensors, its internal model predicts the forward vehicle's position based on its last known speed.
- **Architecture Diagram:**

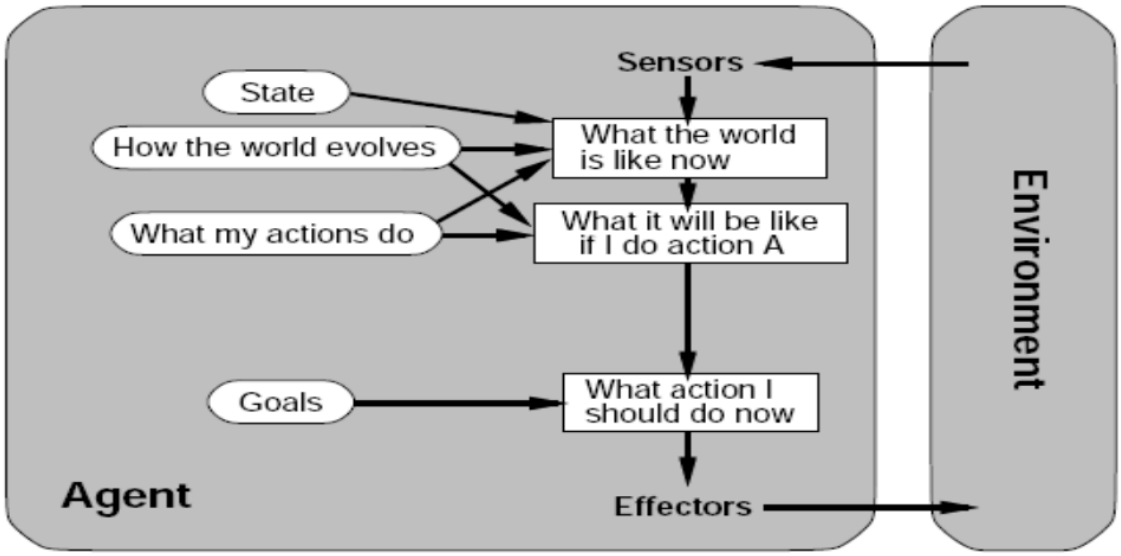



```
function MODEL-BASED-REFLEX-AGENT( percept ) returns an action
  persistent: state, the agent's current conception of the world state
              model , a description of how the next state depends on current state and action
              rules, a set of condition–action rules
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action , percept, model )
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

3. Goal-Based Agent

- **Description:** Combines internal state tracking with explicit **Goal Information** detailing desirable situations. It evaluates future projections by asking: *"What will happen if I take action A?"* This architecture is highly flexible because goal representations are stored explicitly and can be dynamically swapped or updated.
- **Example:** An autonomous taxi with a designated target destination (e.g., Cairo International Airport) choosing routing combinations specifically to satisfy that goal.
- **Architecture Diagram:**



- **Algorithm of goal-based agent**

function GOAL-BASED-AGENT(*percept*) **returns** an action
persistent: *state*, the agent's current conception of the world state
goal, a description of what the agent would like to achieve
rules, a set of condition-action rules
action, the most recent action, initially none

state ← UPDATE-STATE (*state*, ~~*action*~~, *percept*, ~~*goal*~~)

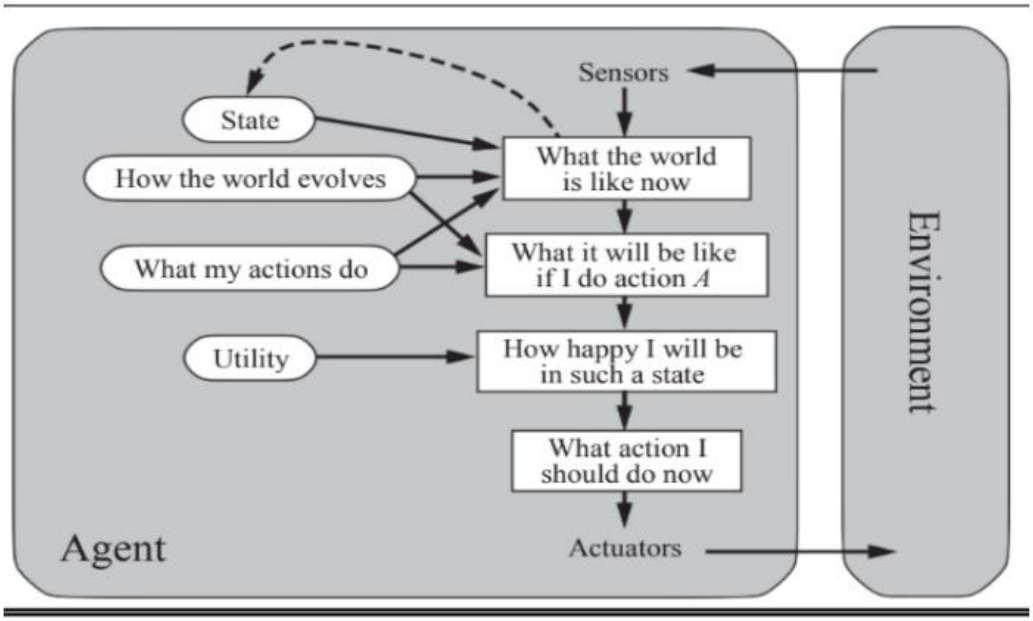
rule ← RULE-MATCH (*state*, *rules*, *goal*)

action ← *rule*.ACTION
state <-- UPDATE-STATE (state, action)

return *action*

4. Utility-Based Agent

- **Description:** Goals alone are often insufficient to distinguish between different paths (e.g., a fast, safe route vs. a narrow, congested route). A **Utility Function** maps states to real numbers to measure "how happy" the agent will be with a given outcome. Under environmental uncertainty, it acts to **maximize expected utility**.
- **Example:** A flight reservation engine sorting routes based on a trade-off between price, duration, and layover counts to optimize traveler preference.
- **Architecture Diagram:**



- **Algorithm of Model-based agent**

function UTILITY-BASED-AGENT(*percept*) **returns** an action
persistent: *state,* the agent's current conception of the world state
possible states, possible states that may maximize happiness
rules, a set of condition-action rules
action, the most recent action, initially none

state ← UPDATE-STATE (*state, action, percept, possible states*)

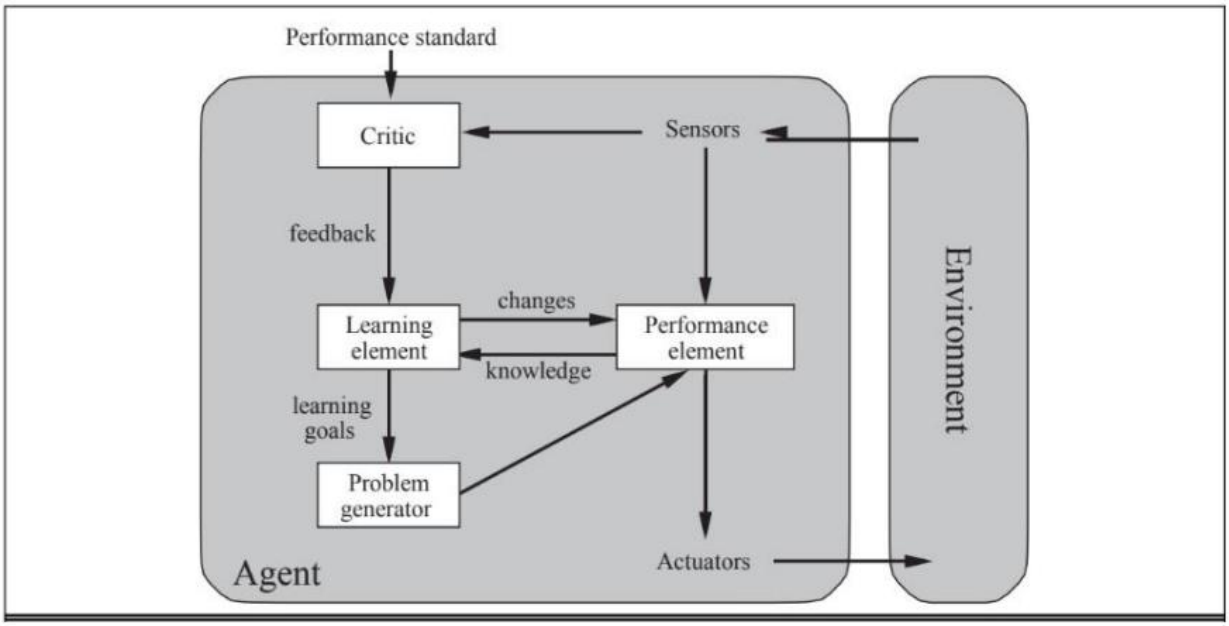
rule ← RULE-MATCH (*state, rules, possible states*)

action ← *rule.*ACTION

return *action*

4. **Learning agents**

✓ All agent types can be turned into learning types



✓ The most important distinction is between the learning element, which is responsible for making improvements, and the performance element, which is responsible for selecting external actions.

✓ The learning element uses feedback from the critic

✓ The last component of the learning agent is the problem generator. It is responsible for suggesting actions that will lead to new and informative experiences.

✓ The performance standard distinguishes part of the incoming percept as a reward (or penalty) that provides direct feedback on the quality of the agent's behavior.

▪ Atomic representation each state of the world is indivisible it has no internal structure.

Section 2: Search Algorithms & State Space

2.1 State Space Representation

State spaces formulate problems using precise mathematical structures, variables, and parameters:

- **Water Jug**
- **N-Queens Puzzle**

- **N-Puzzles (e.g., 8-Puzzle)**

2.2 Search Algorithms Breakdown

Search evaluation relies on four pillars: **Completeness** (guaranteed to find a solution if one exists), **Optimality** (finds the lowest-cost solution), **Time Complexity** (how long it takes), and **Space Complexity** (how much memory it consumes).

1. Depth-First Search (DFS)

- **Strategy:** Expands the deepest node in the current frontier branch first, then utilizes backtracking when a dead end is encountered.
- **Pros:** Highly efficient memory usage; space complexity is linear with depth (O(bm)). Simple to implement recursively via the function call stack.
- **Cons:** Risk of getting trapped in infinite, non-terminating paths (Incomplete in infinite spaces). The discovered path is often non-optimal.

2. Breadth-First Search (BFS)

- **Strategy:** Explores all nodes at the current depth level completely before moving down to the next deeper level.
- **Pros:** Complete (guaranteed to find a goal if it exists) and optimal if all step costs are identical.
- **Cons:** Extremely high memory consumption; space complexity grows exponentially (O(b^d)).

3. Uniform Cost Search (UCS)

- **Strategy:** A generalization of BFS that expands nodes based on the lowest accumulated path cost $g(n)$ from the root node to the current node, utilizing a Priority Queue.
- **Procedural Steps:**
    1. Insert the initial state into the priority queue with a cost of 0.
    2. While the priority queue is not empty, loop:
        - a. Extract the node with the lowest cumulative path cost $g(n)$ from the head of the queue.
        - b. If the extracted node is the goal state, return the node and its path.

        - c. Otherwise, expand the node by generating all its children, computing their cumulative costs, and inserting them into the priority queue.

4. Search Algorithm

- **Core Evaluation Equation:**

$f(n) = g(n) + h(n)$

Where g(n) is the exact accumulated cost to reach node n from the start, and h(n) is the heuristic function estimating the remaining cost to the goal.

2.3 Search Queue Tracing Simulation

Given a starting state S expanded to connected nodes A, B, and C Straight-line heuristic values are defined as: S=7, C=5, B=9, A=10, G=0. Individual step costs are derived from the map.

2.4 Decision Trees & Binary Representation

- **Binary Search/Query Trees:** Highly effective structures for representing AI problems driven by binary **Yes/No** inquiries.
    - **Root Selection Rule:** When mapping out a binary tree for problem resolution, selecting the initial root is a critical step. As a strict rule for exam scenarios: **Take the very first question presented in the problem statement data and designate it as the absolute root of your tree**, then branch out systematically from there.
- **Decision Trees & Random Forests:**
    - Decision tree representations serve as the foundational backbone for ensemble algorithms like **Random Forests**.
    - **Heuristic Root Selection:** In practical scenarios like a clinical application diagnosing patients (the Doctor Example), the root is typically chosen based on prior experience or statistical significance. Place the most critical or highest-entropy symptom—the one that divides options most evenly—as the root. If a later question proves more effective at organizing the classification hierarchy, you can readjust the root during construction.

Section 3: Knowledge Representation & Engineering

3.1 Knowledge Bases & Structuring Mechanics

- **Knowledge Base (KB):** A centralized repository containing facts, rules, and formal representations of a specific domain. Common methodologies include First-Order Logic, Semantic Networks, Frames, and Binary Trees.
- **Object-Attribute-Value Triplets (OAV):**
    - **Definition:** A foundational relational mechanism that binds a specific object to an attribute and assigns its corresponding value.
    - **Example:** Object: Car | Attribute: Color | Value: Red.
    - **Connection to Frames:** OAV triplets provide the high-level structural framework upon which complex frames are constructed.
- **Frames:**
    - **Definition:** A structural method of representing knowledge using a collection of attributes and values that describe a prototypical object.
    - Structurally, a frame consists of two key components: **Slots** and **Fillers**.
    - **Slots:** Structural fields identifying the attributes or characteristics of the entity (e.g., Attributes like: Name, Height, Engine_Type).
    - **Fillers:** The specific data entries or values that populate those slots (e.g., Real Data like: Ahmed, 180cm, V8).
    - **Comparative Rule:** Semantic Networks exhibit flexible predictive and pattern-matching capabilities directly from relational networks, whereas Frames operate as structured, rigid, and predefined tabular schemas.

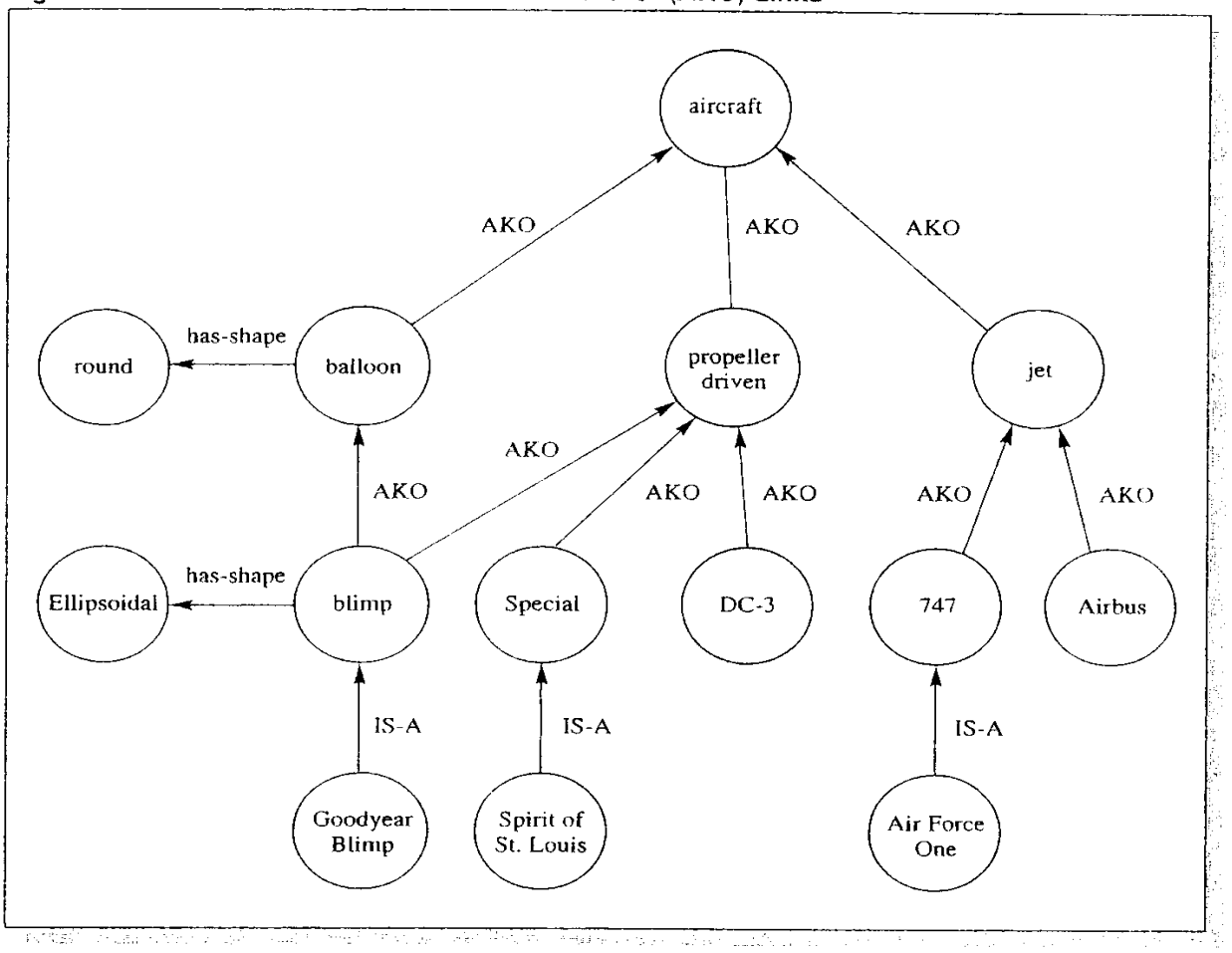
3.2 Semantic Networks

- **Core Graphical Components:** Any formal semantic network visualization must contain:
    1. **Nodes:** Explicitly representing entities, concepts, or objects (Two Explicit Nodes per relationship).
    2. **Links / Arcs:** Directed arrows explicitly defining the exact relationship linking those nodes together.
- **Explicit vs. Implicit Knowledge:**
    - **Explicit Knowledge:** Information stated directly on the graph via an explicit link (e.g., Carrol xrightarrow Is-A Sister of Ann).

- **Implicit Knowledge:** Facts not drawn directly but inferred via property inheritance along the relational hierarchy. For instance, if the graph explicitly shows that a "Sparrow is a bird" and a "Bird can fly", we implicitly infer that a "Sparrow can fly" without requiring an explicit arrow connecting the two.

- **Aircraft Classification Problem:**
  - When organizing taxonomic categories like aircraft, structural relationships are governed by keywords like **Is-A** (denoting individual membership in a category) or **A Kind Of** (denoting sub-category or subclass specialization).
  - **Network Visualization:**

Figure 2.5 A Semantic Net with IS-A and A-Kind-Of (AKO) Links



In the exam, map out all explicit statements into distinct nodes and label every directional arc carefully.

Section 4: Expert Systems & Production Architecture

4.1 Expert Systems vs. Machine Learning

This direct comparison highlights the architectural differences between the two paradigms:

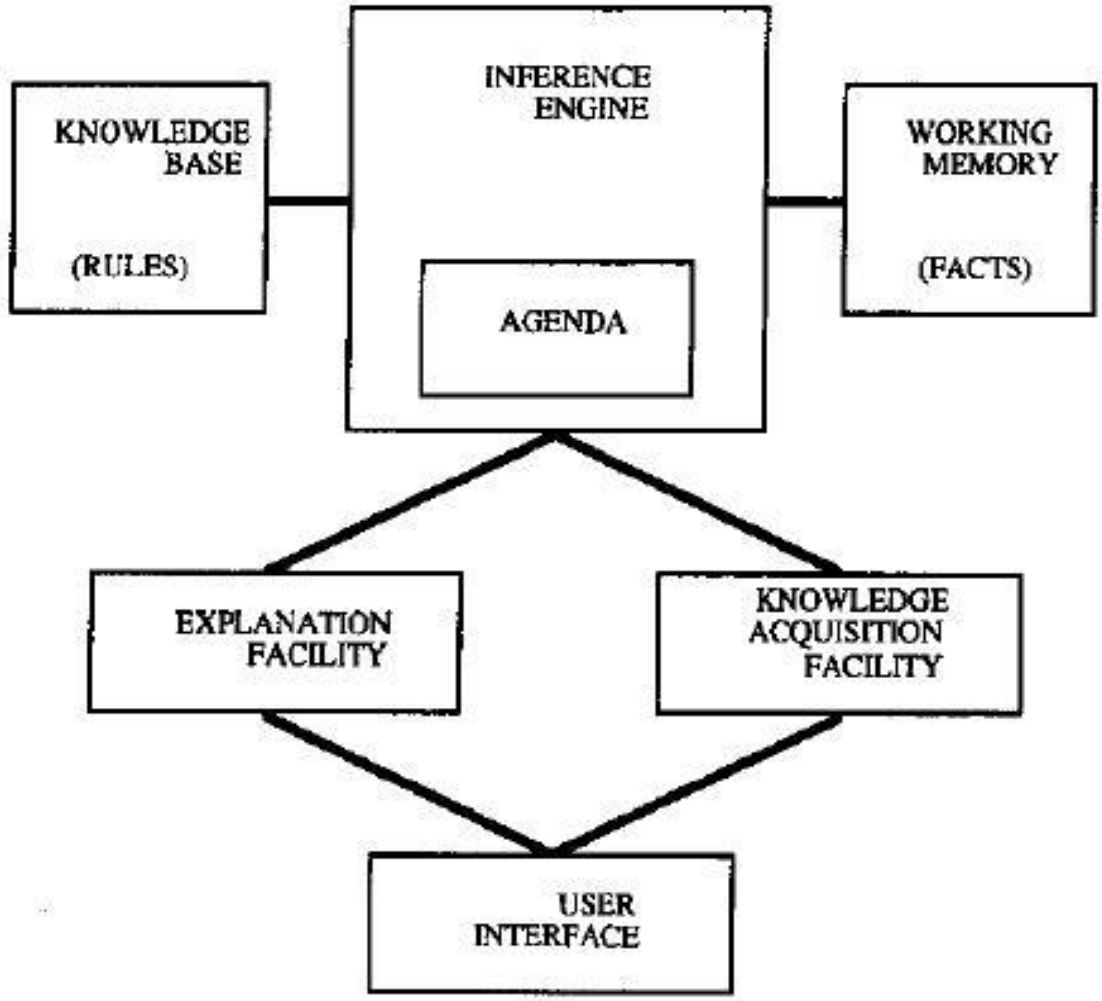
Criteria	Expert Systems (ES)	Machine Learning (ML)
Performance Driver	Dependent entirely on the accuracy and depth of rule sets extracted from human domain experts.	Dependent on the quality, volume, and variance of data alongside structural algorithms.
Core Mechanism	Logical inference, rule traversal, and deterministic "If-Then" logic loops.	Statistical models, mathematical pattern recognition, and loss function minimization.
Production Rules	Contains explicit, human-readable logic rules hardcoded into the software framework.	No explicit human rules; knowledge is distributed across numeric weight matrices.
Working Memory	Contains current active facts, real-time percepts, and an **Agenda** controlling rule execution.	Stores feature vectors, mathematical tensors, and intermediate layer matrices.
Knowledge Base	Clear, explicit representation of human expertise structured by a knowledge engineer.	Hidden implicitly within statistical parameters, weights, and mathematical biases.
Learning Ability	Static; cannot adapt or generate new rules automatically. Requires manual updates.	Dynamic; learns and improves continuously

Criteria	Expert Systems (ES)	Machine Learning (ML)
		through performance evaluation loops.

4.2 Expert System Blueprints (With & Without Eclipse)

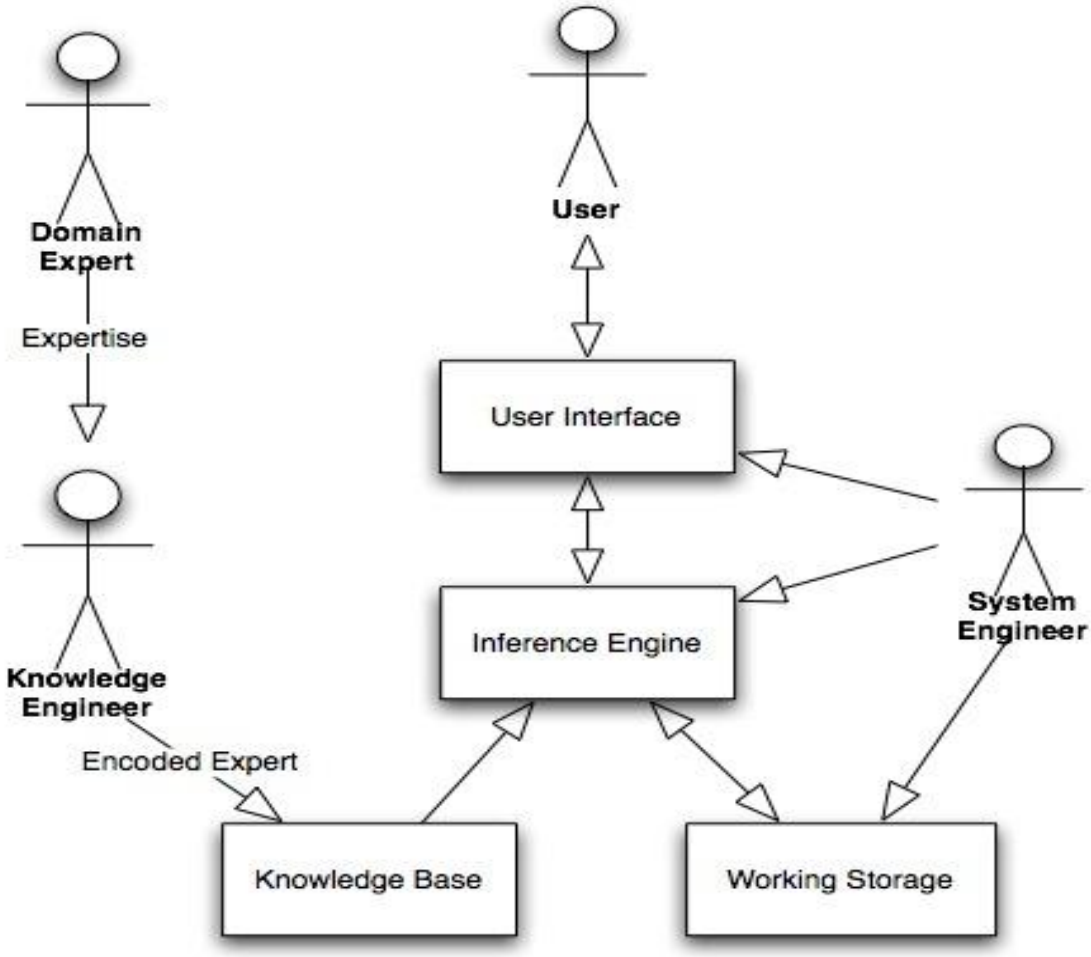
Scenario A: Structural Design Managed Within Eclipse Environment

- **Description:** This blueprint includes internal software modules integrated with an execution manager. The key element is the **Agenda**, a dynamic scheduling array that maintains conflict sets of activated rules waiting to be fired based on current priorities.
- **Architecture Block Diagram:**



Scenario B: Standard Architecture Without Eclipse Environment

- **Description:** Focuses on the classical, interactive interface loop, highlighting how an external actor interacts with the core engine.
- **Architecture Block Diagram:**



- **Engine Outputs:** The final diagnosis, conclusion, or analytical decision returned to the interface after the inference engine runs all matching rules against working memory.
- **Syllabus Exclusion Note:** The terms **Language**, **Shell**, and **Tools** from earlier Eclipse reference discussions are excluded from this exam. Do not include them in your answers.

Section 5: Natural Language Processing (NLP)

5.1 Context-Free Grammar (CFG) & Parsing Tree Synthesis

- **Context-Free Grammar (CFG):** A formal set of structural production rules that define how high-level sentence components (such as a Noun Phrase NP or Verb Phrase VP) expand into structural terminal words.
- **Parsing:** The computational process of breaking down a string of words against target CFG rules to output a hierarchical syntax tree detailing the grammatical role of each word.
- **NLTK Parsing Execution Code:**

The following program builds a CFG structure and parses a target sentence to output brackets representing the parse tree.

Python

```
import nltk
from nltk.tokenize import word_tokenize


# Define strict context-free grammar constraints
grammar = nltk.CFG.fromstring("""
  S -> NP VP
  NP -> NNP NNS | NN NNS | CD NN | NP PP | Det N | N N | N
  VP -> V NP | V NP PP | V VP
  PP -> P NP
  NNP -> 'Fed'
  V -> 'raises' | 'strikes' | 'put'
```

```
  NN -> 'interest' | 'effort' | 'control' | 'inflation' | 'student' | 'book' | 'table'
  NNS -> 'rates' | 'kids'
  CD -> '0.5'
  P -> '%' | 'in' | 'to' | 'on'
  Det -> 'the'
  N -> 'Teacher' | 'student' | 'book' | 'table'
  A -> 'idle'
""")

# Target input sequence
sentence = "the student put the book on the table"
tokens = word_tokenize(sentence)

# Initialize chart parser and extract structural groupings
parser = nltk.ChartParser(grammar)
for tree in parser.parse(tokens):
  print(tree)
  # Target bracketed output to write down in the exam sheet:
  # (S (NP (Det the) (N student)) (VP (V put) (NP (Det the) (N book)) (PP (P on) (NP (Det the) (N table)))))
```

- **Grammar Scope Note:** The exam emphasizes **Deterministic Finite Automata (DFA)** state machines to evaluate text acceptance and basic morphology patterns, rather than deep CFG tree parsing structures.

Section 6: Comprehensive Practice Scenarios

6.1 Knowledge Base Problem Matching & Conflict Resolution

Problem Selection Matrix

- **Binary Trees:** Best suited for deterministic classification problems driven by quick binary choices (e.g., diagnosing simple consumer electronic hardware failures using Yes/No choices).
- **Semantic Networks:** Ideal for hierarchical domain knowledge with complex inheritance patterns (e.g., biological taxonomy systems or comprehensive aircraft categorization schemas).
- **Frames:** Most effective for structural database records or object-oriented structures requiring detailed, standardized property layouts (e.g., tracking structural properties in real estate or room configurations).
- **First-Order Logic:** Best for strict formal verification, mathematical theorem proving, and multi-conditional policy compliance systems.

The Seven Conflict Resolution Tactics in Eclipse (With Examples)

1. **Salience:** Assigns hardcoded priority weights to rules.
    - *Example:* A rule for a patient's life-saving treatment is assigned salience 100, while a rule for billing paperwork is assigned salience 10. The life-saving treatment fires first.
2. **Depth (LIFO):** Prioritizes the rule activated by the most recently added fact in working memory.
    - *Example:* If an emergency alarm fact enters working memory, the system pauses older tasks to handle the sudden alarm immediately.
3. **Breadth (FIFO):** Executes rules based on their order of activation, handling the oldest activations first.
    - *Example:* A standard customer service ticket queue; the ticket submitted first is processed before later arrivals.
4. **Simplicity:** Prioritizes rules containing fewer conditional clauses (antecedents) in their IF statement.
    - *Example:* A rule stating IF (temp > 40) -> TurnOnFan fires before a more complex rule stating IF (temp > 40) AND (humidity > 80) AND (daytime) -> TurnOnAC.
5. **Complexity:** Prioritizes highly specific rules containing a larger number of conditional checks.

    - *Example:* In failure diagnostics, a rule checking 5 specific sensor errors at once takes priority over a general rule checking only one.
6. **Random:** Selects a rule from the conflict set entirely at random when priorities are equal.
    - *Example:* Distributing identical background data maintenance tasks evenly among available server threads.
7. **Specificity:** Prioritizes rules that look for explicit constant values over rules that match generic variable parameters.
    - *Example:* A rule that triggers specifically for a Mercedes 2026 Model fires before a general rule matching Any German Vehicle.

Lecture: Definite Clause Grammar (DCG)

1. What is DCG?

Definite Clause Grammar (DCG) is a formal grammar notation used in Prolog for describing natural language syntax.

2. Why DCG?

- Easier than writing parsing rules manually.
- Integrated with Prolog inference engine.
- Used for Natural Language Processing.
- Automatically generates parsers.

3. DCG Example

Sentence:

the boy eats apple

Grammar:

sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.

verb_phrase --> verb, noun.

```
determiner --> [the].

noun --> [boy].

noun --> [apple].

verb --> [eats].
```

4. Advantages of DCG

1. Easy to write.
2. Easy to understand.
3. Works directly with Prolog.
4. Suitable for NLP.

5. Difference Between DCG and CFG

DCG	CFG
Implemented in Prolog	Theoretical Grammar
Executable	Description only
Used in NLP systems	Used in parsing theory
Supports logic programming	Does not

Complete

What is DCG?

Answer:

A grammar notation used in Prolog to represent natural language syntax.

Why is DCG better than CFG?

Answer:

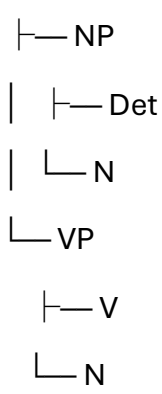
Because DCG can be executed directly inside Prolog and used for parsing.

Parse Tree

What is Parse Tree?

A tree representation showing the grammatical structure of a sentence.

Example:



Sentence:

The boy eats apple

Ambiguity

Syntactic Ambiguity

More than one parse tree.

Example:

I saw a man with a telescope.

Possible meanings:

1. I used a telescope.
2. The man had a telescope.
3.

Semantic Ambiguity

Sentence structure is clear.

Meaning is unclear.

Example:

Bank

Could mean:

- River bank
- Financial bank

NLTK Parse Tree

```
import nltk

grammar = nltk.CFG.fromstring("""
S -> NP VP
NP -> Det N
VP -> V N
Det -> 'the'
N -> 'boy' | 'apple'
V -> 'eats'
""")

parser = nltk.ChartParser(grammar)

sentence = "the boy eats apple".split()

for tree in parser.parse(sentence):
   print(tree)
```

Zero Shot Learning

What is Zero Shot Learning?

A machine learning technique where a model predicts classes it has never seen during training.

Example:

Training:

Cat

Dog

Horse

Test:

Tiger

Model predicts:

Tiger belongs to animal class

without seeing tiger before.

Advantages

- No retraining needed.
- Handles unseen classes.
- Useful in NLP.

Example

ChatGPT:

Translate this sentence to Arabic.

Even if sentence style never appeared in training.

Flowise

What is Flowise?

Flowise is a visual AI workflow builder for LLM applications.

Users create AI systems by drag and drop nodes.

Components

Input Node

Receives user question.

LLM Node

ChatGPT, Gemini, Claude.

Memory Node

Stores conversation history.

Output Node

Returns final answer.

Advantages

1. No coding required.
2. Fast development.
3. Visual representation.
4. Integrates with LangChain.

Complete

What is Flowise?

Answer:

Visual tool for building AI applications using drag and drop workflows.

Knowledge Engineering Process

Steps

1. Knowledge Acquisition

Collect knowledge from experts.

2. Knowledge Representation

Store knowledge.

Examples:

- Frames
- Semantic Nets
- Rules

3. Knowledge Validation

Verify correctness.

4. Knowledge Inference

Generate conclusions.

5. Knowledge Maintenance

Update knowledge base.

OAV Triplet

OAV

Object – Attribute – Value

Example:

Car

Color

Red

Object:

Car

Attribute:

Color

Value:

Red

Complete

What is OAV?

Answer:

Association between Object, Attribute and Value.

Frame

What is Frame?

A knowledge representation structure based on slots and fillers.

Example

Student

Name = Ahmed

Age = 21

Department = CS

Slots

Attributes.

Example:

Age

Name

Department

Fillers

Actual values.

Example:

21

Ahmed

CS

Forward Chaining

Definition

Data Driven Reasoning.

Starts from facts.

Moves toward conclusion.

Example:

Fact:

Engine not start

Rule:

IF engine not start

THEN battery problem

Result:

Battery Problem

Backward Chaining

Definition

Goal Driven Reasoning.

Starts from goal.

Moves backward to facts.

Example:

Goal:

Battery Problem?

Check rule.

Check facts.

Then conclude.

Expert System Car Diagnosis

Facts

Car not starting

Lights weak

Rule

IF lights weak

AND car not start

THEN battery problem

Conclusion

Battery problem

Shoulder Diagnosis Example

IF shoulder pain

AND movement limitation

THEN frozen shoulder

IF shoulder pain

AND swelling

THEN tendon injury

Conflict Resolution (المهم جدًا)

داخل Expert System لو أكتر من Rule اتحققت.

مين يتنفذ؟

7 أشهر Methods

1. Priority

نفذ الأعلى أولوية.

2. Specificity

نفذ الأكثر تحديدًا.

3. Recency

اعتمد على أحدث Facts.

4. Refractoriness

متنفذش نفس Rule مرتين.

5. Random

اختيار عشوائي.

6. First Applicable

أول Rule اتحققت.

7. Context Limitation

اختيار حسب Context.

Semantic Network

Definition

Graph representation of knowledge using:

- Nodes

- Links

Example

Canary IS-A Bird

Bird IS-A Animal

Canary

|

IS-A

|

Bird

|

IS-A

|

Animal

Explicit Knowledge

Stored directly.

Example:

Bird has wings.

Implicit Knowledge

Derived.

Example:

Canary has wings.

because

Canary IS-A Bird