UNIT III: KNOWLEDGE REPRESENTATION using Predicate Logic and Rules (using Prolog Programming Language)

(it's the foundation of Reasoning and Decision-making in AI)
(It's all about making AI systems that don't just compute, but also reason like humans)

Contents

Section-1: Logical Foundations of Knowledge Representation and Inference

Propositional Logic

Definition

Propositional Logic is a system of logic that uses complete statements (called "propositions") that can only be TRUE or FALSE, and shows how to combine them using words like AND, OR, and IF...THEN to form more complex statements.

"Proposition" = A statement or claim that can be true or false
"Logic" = The study of correct reasoning

So **"Propositional Logic"** literally means: **"The logic of statements that are true or false."**

Where "Proposition" Comes From

The word comes from the idea of "proposing" or "stating" something. For example:

- "It is raining" - This is a **proposition** because you're stating something that can be verified as true or false.
- "The cat is on the mat" - Another **proposition**.
- "2 + 2 = 4" - Also a **proposition** (true in this case).

What's NOT a Proposition?

- **Questions:** "Is it raining?" (Can't be true/false - it's asking)
- **Commands:** "Close the door!" (Not true/false - it's directing)
- **Opinions:** "Pizza is delicious" (Debatable, not clearly true/false in logic)

Note: So, the name tells you exactly what it is: **The logic system that works with true/false statements (propositions).**

Let's understand it better with an example – The Game Rule Cards

Think of it like: You have a deck of true/false cards that describe your game character's status. Each card is either **FACE-UP (True)** or **FACE-DOWN (False)**.

- `A = "Character has a Key"`
- `B = "Character is at the Door"`
- `C = "Character has a Sword"`
- `D = "Monster is nearby"`

Each of these is a simple **Proposition** - it's either true right now or it's false.

The Rule Maker (Connectives):

These let you combine the cards to create game rules!

- **AND (∧):** The "COMBO MOVE" rule. **Both** conditions must be true to perform the action.
  - `B AND A` → "Character is at the Door **AND** has a Key" = **OPEN DOOR**
  - If you're at the door but don't have the key, you can't open it. If you have the key but aren't at a door, nothing happens. You need **both**.
- **OR (∨):** The "BACKUP PLAN" rule. **At least one** condition must be true.
  - `C OR D` → "Character has a Sword **OR** a Monster is nearby" = **CHARACTER IS NERVOUS** (maybe their hand shakes on screen).
  - This is true if just one of these things is happening, or if both are happening.
- **NOT (¬):** The "OPPOSITE" rule. It flips the card over.
  - `NOT C` means "Character does **NOT** have a Sword."
  - If `C` was Face-Up (True), then `NOT C` is Face-Down (False).
- **IMPLIES (→):** The "IF/THEN" rule. It creates a cause-and-effect chain.
  - `D → C` means "**IF** a Monster is nearby, **THEN** the Character has a Sword."
  - This is the game's way of automatically giving you a sword when a monster appears! If a monster shows up (`D` is True) but you don't get a sword (`C` is False), then the game rule has been broken.

Let's Play the Game!

Current Game State:

- `A = True` (You HAVE the key)
- `B = False` (You are NOT at a door)
- `C = True` (You HAVE a sword)
- `D = False` (No monster is nearby)

Now, let's check our rules:

1. `B AND A` ("Open Door") → **False** (because `B` is False! You're not at a door.)
2. `C OR D` ("Character is Nervous") → **True** (because `C` is True! You have a sword, so you're ready for a fight, which makes you nervous.)
3. `NOT D` → **True** (The opposite of `D`. Since `D` is False "No monster", `NOT D` means "It is NOT true that no monster is nearby"... wait, that's confusing! Let's simplify: `NOT D` really means "It is safe." And yes, it is safe!)
4. `D → C` ("Monsters make swords appear") → **True** (Why? Because there is no monster (`D` is False). The "If" part didn't happen, so the rule wasn't triggered. The promise wasn't broken.)

The Limitation:

What if we want a rule that says "ALL monsters are dangerous"? In our current system, we'd need a separate card for *every single monster*: `Monster1_is_nearby`, `Monster2_is_nearby`... That would be a huge, messy deck of cards!

That's why we need to upgrade to **Predicate Logic** for more complex games, where we can have rules about "all" things or "some" things.

But for simple, yes/no situations, Propositional Logic is your perfect set of game rule cards!

Predicate Logic

Definition

Predicate Logic is a system of logic that lets us talk about the PROPERTIES of things and the RELATIONSHIPS between things, using placeholders that can represent many different objects.

Think of it like: **Template Logic** - it uses sentence templates with blanks that you can fill in.

It breaks down statements into two parts: the **subject** (what we are talking about) and the **predicate** (what we are saying about the subject). It also uses special words like "all" and "some" to talk about entire groups or at least one thing.

Why the Name?

The word "predicate" comes from the part of the sentence that says something about the subject. In the sentence "The sky is blue," "is blue" is the predicate. So, Predicate Logic is the logic that uses predicates to describe the properties of and relationships between objects.

Example

A detective's form with blanks to fill in. Instead of whole sentences, we have templates.

- A **Predicate** is the template. It describes a property or a relationship.
  - `IsHungry(x)` - This isn't True or False yet. It means "x is hungry." We need to fill in the blank!
- A **Constant** is a specific thing you put in the blank.
  - `Dad`, `Mom`, `The_Pepperoni_Pizza`
- Now we can make facts:
  - `IsHungry(Dad)` - "Dad is hungry." (A solid fact, like in Propositional Logic).
  - `Loves(Mom, Pizza)` - "Mom loves Pizza." (This shows a relationship between two things).

The Superpowers (Quantifiers):

This is the magic that Propositional Logic didn't have.

- **∀ (For All):** The "Every Single One" magnifying glass.
  - `∀x IsHungry(x)` would mean "**For all** x, x is hungry." Or, "**Everyone** is hungry." This is probably False.
- **∃ (There Exists):** The "At Least One" magnifying glass.
  - `∃x IsHungry(x)` means "**There exists** an x who is hungry." Or, "**Someone** is hungry." This is much more likely to be True!

Let's Build a Rule for Pizza Night:

1. "All people who are hungry get pizza."
   - In Detective Notebook language: `∀x ( IsHungry(x) → GetsPizza(x) )`
   - Translation: "For every person x, **IF** x is hungry, **THEN** x gets pizza."
2. Let's add a fact: `IsHungry(Dad)` - "Dad is hungry."

Our robot brain can now figure out: Since the rule applies to **all** `x`, and `Dad` is one of them, and he *is* hungry, then he must get pizza! `GetsPizza(Dad)` must be **True**.

Unification & Resolution algorithm

These two are **core algorithms** in *Rule-Based Artificial Intelligence* and *Automated Reasoning* (especially when using **Predicate Logic** as Knowledge Representation).

Unification and Resolution are two different algorithms, but **they work together** in reasoning systems like **Prolog** or **AI Rule-based inference engines**.

Unification is the process of **making two logical expressions identical** by finding a suitable substitution of variables.

It's heavily used in **Prolog**, **Resolution algorithm**, and **Expert Systems** to check whether two predicates can match.

Prolog automatically performs unification every time it tries to satisfy a goal.

Prolog uses the **Resolution algorithm** (specifically *SLD Resolution — Selective Linear Definite clause* resolution) to answer queries.

Note: We shall see examples in the upcoming Prolog Programming Language section.

We'll use this **knowledge base**:

```prolog
likes(ram, apples).
likes(sita, bananas).
likes(gita, X) :- likes(ram, X).
```

And the **query**:

```prolog
?- likes(gita, What).
```

The Unification Algorithm (Step-by-Step)

Step 1: Check the Structure

- **Action:** Compare the main functor (the predicate/function name) and the number of arguments (arity) of the two expressions.
- **Logic:** If the functors are different (likes vs hates) or the number of arguments don't match, unification **fails immediately**.
- **Our Example:**
  - likes(ram, X) & likes(Y, apples)
  - Functor: likes = likes ✅
  - Arity: 2 arguments = 2 arguments ✅
  - **Result:** Proceed to the next step.

Step 2: Handle Constants

- **Action:** If both expressions are constants (or both arguments being compared are constants), they must be identical.
- **Logic:** Constants are fixed values. You can't substitute for them.
- **Your Example:**
  - likes(ram, apples) & likes(ram, apples)
  - ram = ram ✅
  - apples = apples ✅
  - **Result:** Unification succeeds with an empty substitution, θ = {}.

Step 3: Handle a Variable and a Term (The Heart of Unification)

This is where the actual substitutions happen.

- **Action:** If one of the expressions is a variable and the other is any term (constant, variable, or compound expression).
- **Logic:** Bind the variable to the term. This is recorded in the substitution set θ.
- **Your Example 1:**
  - likes(ram, X) & likes(ram, apples)
  - Comparing X (a variable) and apples (a constant).
  - **Action:** Bind X to apples.
  - **Result:** θ = { X / apples }

Note: θ : Indicates substitution/replace

- **Important Check (The "Occurs-Check"):** A smart algorithm also performs an "occurs-check" to prevent a variable from being bound to a term that contains itself (e.g., X = f(X)). This would create an infinite structure. Your table simplifies this, but it's a critical part of a robust implementation.

Step 4: Handle Compound Terms (Recursion)

- **Action:** If both expressions are compound terms (like likes(gita, X)), you must recursively unify their corresponding arguments.
- **Logic:** The algorithm breaks down the problem into smaller sub-problems.
- **Your Example:**
  - likes(gita, X) & likes(Y, apples)
  1. Unify the first arguments: gita and Y.
     - Y is a variable, gita is a constant.
     - **Substitution:** θ_1 = { Y / gita }
  2. *Now, apply this substitution to the rest of the expressions.* The expressions become: likes(gita, X) and likes(gita, apples).
  3. Unify the next arguments: X and apples.
     - X is a variable, apples is a constant.
     - **Substitution:** θ_2 = { X / apples }

Step 5: Combine Substitutions

- **Action:** Collect all the substitutions from the recursive unification steps and combine them into a single Most General Unifier (MGU).
- **Logic:** The MGU is the cumulative effect of all the bindings.
- **Your Example (from Step 4):**
  - We have θ_1 = { Y / gita } and θ_2 = { X / apples }.
  - **Combine them:** MGU = { Y / gita, X / apples }
  - Applying this MGU to both original expressions gives us: likes(gita, apples) and likes(gita, apples). They are now identical! ✅

Step 6: Handle Conflicts

- **Action:** If at any point a conflict is found, unification fails.
- **Logic:** A conflict arises when two constants are different, or when a variable would have to be bound to two different values.
- **Your Example:**
  - likes(ram, X) & likes(sita, bananas)
  1. Unify the first arguments: ram and sita.
     - Both are constants, but ram ≠ sita.
     - **Result:** ❌ Fail immediately. There is no need to check the second argument.

First-Order Logic

Note: The resolution algorithm requires all statements to be in **Clause Form (Conjunctive Normal Form)**, which means: Only AND (∧) and OR (∨) operations.

So **¬Human(x) ∨ Mortal(x)** is essentially the logical way to say "humans are mortal" in a form that computers can process systematically using resolution.

First-Order Logic (FOL) is a formal system for representing and reasoning about knowledge. It's like a "language of logic" that computers can understand and process systematically.

Quantifiers - Scope Statements

Universal Quantifier (∀) - "For all"

- ∀x Human(x) → Mortal(x)
- **Meaning:** "For every x, if x is human, then x is mortal"
- **Translation:** "All humans are mortal"

Existential Quantifier (∃) - "There exists"

- ∃x Human(x) ∧ Likes(x, apples)
- **Meaning:** "There exists an x such that x is human and x likes apples"
- **Translation:** "Some human likes apples"

Propositional Logic Vs First-Order Logic

Propositional Logic = Simple sentences like "The cat is on the mat"
First-Order Logic = Can say "All cats are on some mat" or "There exists a cat that is black"

Resolution Algorithm Steps

Example Problem
Let's use this knowledge base:

1. "All humans are mortal"
2. "Socrates is a human"
3. "If someone is mortal, they will die someday"

We want to prove: "Socrates will die someday"

1. **Convert all statements to clause form (CNF).**

```
Convert all statements to clause form (CNF)
First, convert to First-Order Logic:

        ∀x: Human(x) → Mortal(x)
        Human(socrates)
        ∀y: Mortal(y) → Die(y)

Convert to Clausal Form (CNF):

        Rule 1: ¬Human(x) ∨ Mortal(x)
        Rule 2: Human(socrates)
        Rule 3: ¬Mortal(y) ∨ Die(y)
```

2. **Negate the goal** (what you want to prove) and add it to the knowledge base.

Our Goal: `Die(socrates)`

Negated Goal: `¬Die(socrates)`

Add to Knowledge Base:

```
C1: ¬Human(x) ∨ Mortal(x)
C2: Human(socrates)
C3: ¬Mortal(y) ∨ Die(y)
C4: ¬Die(socrates)    ← Negated Goal
```

3. **Find two clauses** that contain **complementary literals** (one positive, one negative).

   Look for one positive and one negative version of the same predicate.
   Found: C3 has Die(y) (positive) and C4 has ¬Die(socrates) (negative)

4. **Unify** them (make the literals identical).

   Unify: `Die(y)` from C3 with `Die(socrates)` from C4

   Unification: They unify with substitution `{y/socrates}`

   Apply substitution to C3:

   - Original C3: `¬Mortal(y) ∨ Die(y)`
   - After substitution: `¬Mortal(socrates) ∨ Die(socrates)`

5. **Resolve** → remove the complementary pair and **combine the rest** into a new clause.

   New Clause (C5): ¬Mortal(socrates)
   Updated Clause Set:

```
C1: ¬Human(x) ∨ Mortal(x)
C2: Human(socrates)
C3: ¬Mortal(y) ∨ Die(y)
C4: ¬Die(socrates)
C5: ¬Mortal(socrates)    ← New from resolution
```

6. **Repeat** until either:
   - You get an **empty clause (□)** → ✅ proved, or
   - No new clauses → ❌ not provable.

Next iteration - Find complementary literals:

- `C1` has `Mortal(x)` (positive)
- `C5` has `¬Mortal(socrates)` (negative)

Unify: `Mortal(x)` with `Mortal(socrates)` → `{x/socrates}`

Apply substitution to C1:

- Original C1: `¬Human(x) ∨ Mortal(x)`
- After substitution: `¬Human(socrates) ∨ Mortal(socrates)`

Resolve C1 and C5:

- C1: `¬Human(socrates) ∨ Mortal(socrates)`
- C5: `¬Mortal(socrates)`

Remove complementary literals `Mortal(socrates)` and `¬Mortal(socrates)`

New Clause (C6): `¬Human(socrates)`

Updated Clause Set:

```
C1: ¬Human(x) ∨ Mortal(x)
C2: Human(socrates)
C3: ¬Mortal(y) ∨ Die(y)
C4: ¬Die(socrates)
C5: ¬Mortal(socrates)
C6: ¬Human(socrates)    ← New from resolution
```

(continued): Repeat again

Find complementary literals:

- `C2` has `Human(socrates)` (positive)
- `C6` has `¬Human(socrates)` (negative)

Resolve C2 and C6:

- C2: `Human(socrates)`
- C6: `¬Human(socrates)`

Remove complementary literals `Human(socrates)` and `¬Human(socrates)`

New Clause (C7): `□` ← **EMPTY CLAUSE!**

Final Result

We derived the **empty clause** □, which means we found a contradiction.
Conclusion: Since assuming the negation of our goal led to a contradiction, our original goal Die(socrates) must be **TRUE**.
Proof complete! Socrates will die someday.

```
Knowledge Base & Negated Goal:
1. { likes(ram, apples) }
2. { likes(sita, bananas) }
3. { ¬likes(X, Z), ¬likes(Y, Z), friend(X, Y) }
4. { ¬friend(ram, sita) }   <-- Start here

            | Resolve (3 & 4) with θ={X/ram, Y/sita}
            V
5. { ¬likes(ram, Z), ¬likes(sita, Z) }

            | Resolve (5 & 1) with θ={Z/apples}
            V
6. { ¬likes(sita, apples) }   <-- STUCK! Can't resolve with Clause 2.

            ❸
            ❌ FAIL. Query is false.
```

So, the query “whether “ram” and “sita” are friends is false

Note: The symbol **"¬"** is the standard logical symbol for **negation**, meaning **"NOT"**.

So to summarize:

- ✅ **We KNOW** likes(sita, bananas) is TRUE
- ❓ **We DON'T KNOW** if likes(sita, apples) is true or false
- ❌ **We CANNOT PROVE** friend(ram, sita) with our current knowledge

In Simple Words

- **Unification** = "Matching names and filling blanks"
- **Resolution** = "Using those matches to reason and find new truths"

Section-2: Prolog Programming Language

What is Prolog

Prolog is a logic programming language that is well-suited for developing **logic-based artificial intelligence applications.**

Prolog or **PRO**gramming in **LOG**ics is a **logical** and **declarative** programming language.

This is particularly suitable for programs that involve **symbolic** or **non-numeric computation**. This is the main reason to use Prolog as the programming language in **Artificial Intelligence**, where **symbol manipulation** and **inference manipulation** are the fundamental tasks.

Developers can **set rules and facts** around a problem, and then Prolog's interpreter will use that information to **automatically infer solutions**.

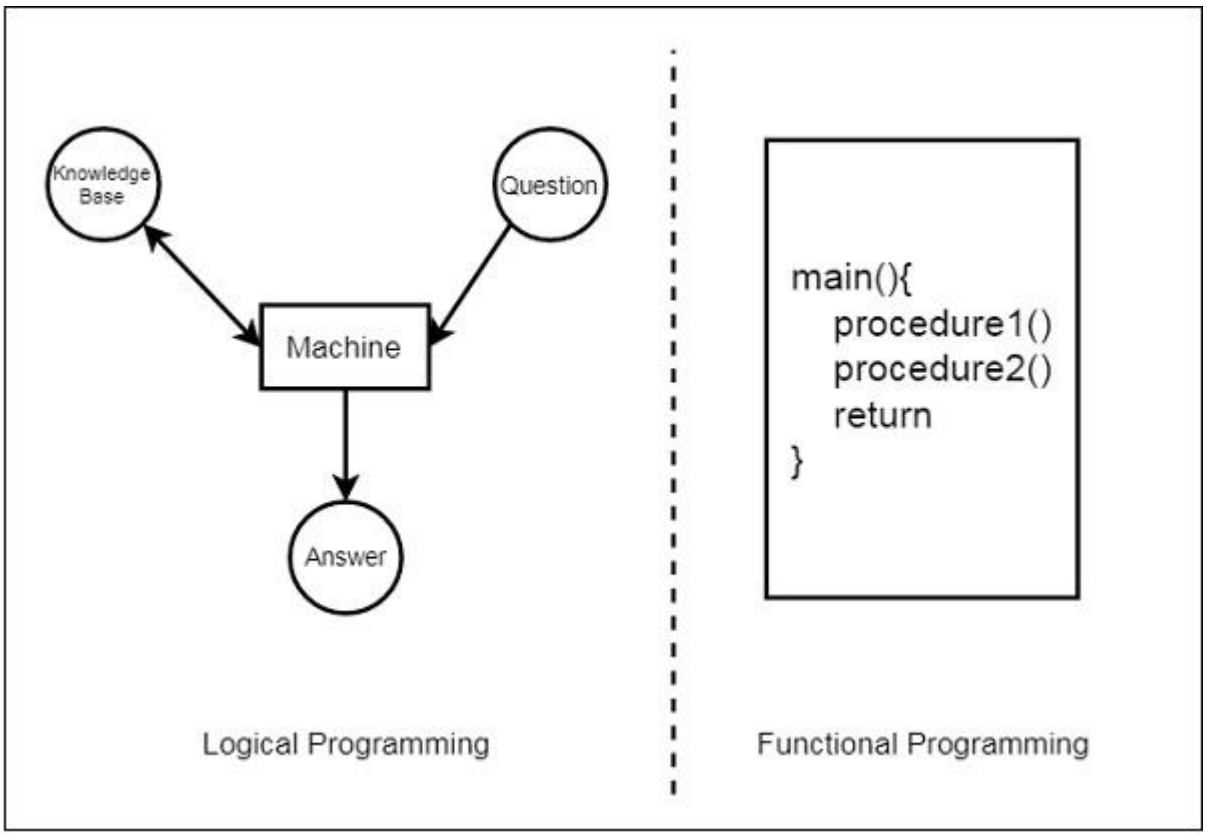
A declarative language tells a computer what it should do, and focuses on the goal itself without providing steps to get to the goal. In contrast, an imperative language tells a computer how to reach a goal step-by-step.

Applications of Prolog

Prolog is used in various domains. It plays a vital role in automation system. Following are some other important fields where Prolog is used:

- Intelligent Database Retrieval
- Natural Language Understanding
- Specification Language
- Machine Learning
- Robot Planning
- Automation System
- Problem Solving

Logic Vs Procedural Programming Languages



Basic building blocks / elements in Prolog

- Facts
- Rules
- Variables
- Questions/Queries

Note: Prolog programs are written using a syntax that is similar to natural language.

Facts

Facts are basic assertions about your world - they're always true.

```prolog
% Syntax: predicate_name(arg1, arg2, ..., argN).
parent(john, mary).      % John is a parent of Mary
parent(susan, mary).     % Susan is a parent of Mary
male(john).              % John is male
female(susan).           % Susan is female
age(john, 35).           % John is 35 years old
```

Think of facts like database records or constant declarations.

Rules

Rules define relationships between facts. They use the `:-` operator which means "if".

```prolog
% Syntax: conclusion :- condition1, condition2, ..., conditionN.

% Rule: X is the father of Y if X is parent of Y and X is male
father(X, Y) :- parent(X, Y), male(X).

% Rule: X is the mother of Y if X is parent of Y and X is female
mother(X, Y) :- parent(X, Y), female(X).

% Rule: X and Y are siblings if they share at least one parent
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
```

Note: X \= Y in Prolog means **"X is not equal to Y"** or "X cannot be unified with Y".

Relationships

In Prolog, **every line that defines a predicate** (whether fact or rule) is establishing a **relationship** between entities.

Why they're ALL called "relationships":

- **Facts**: Direct, explicit relationships (`john is parent of mary`)
- **Rules**: Derived, logical relationships (`X is father of Y if X is parent and male`)

Variables

Variables start with uppercase letters or underscores. They represent to find.

```prolog
% Examples of variables:
X          % A single variable
Person     % Another variable
_          % Anonymous variable (don't care about this value)
_Result    % Another anonymous variable
```

Questions/Queries

Queries are questions you ask Prolog to answer based on your facts and rules.

```prolog
% Query: "Who is a parent of Mary?"
?- parent(X, mary).

% Prolog responds:
X = john ;
X = susan

% Query: "Is John male?"
?- male(john).
true

% Query: "How old is John?"
?- age(john, Age).
Age = 35
```

In the first query:

- father(X, mary) - This is the query pattern
- X - This is an **unbound variable** (unknown value that Prolog will try to find)
- mary - This is a **bound value** (known constant we're searching for)
- ?- - This indicates it's a query (question)

So,

?- father(X, mary). means exactly:

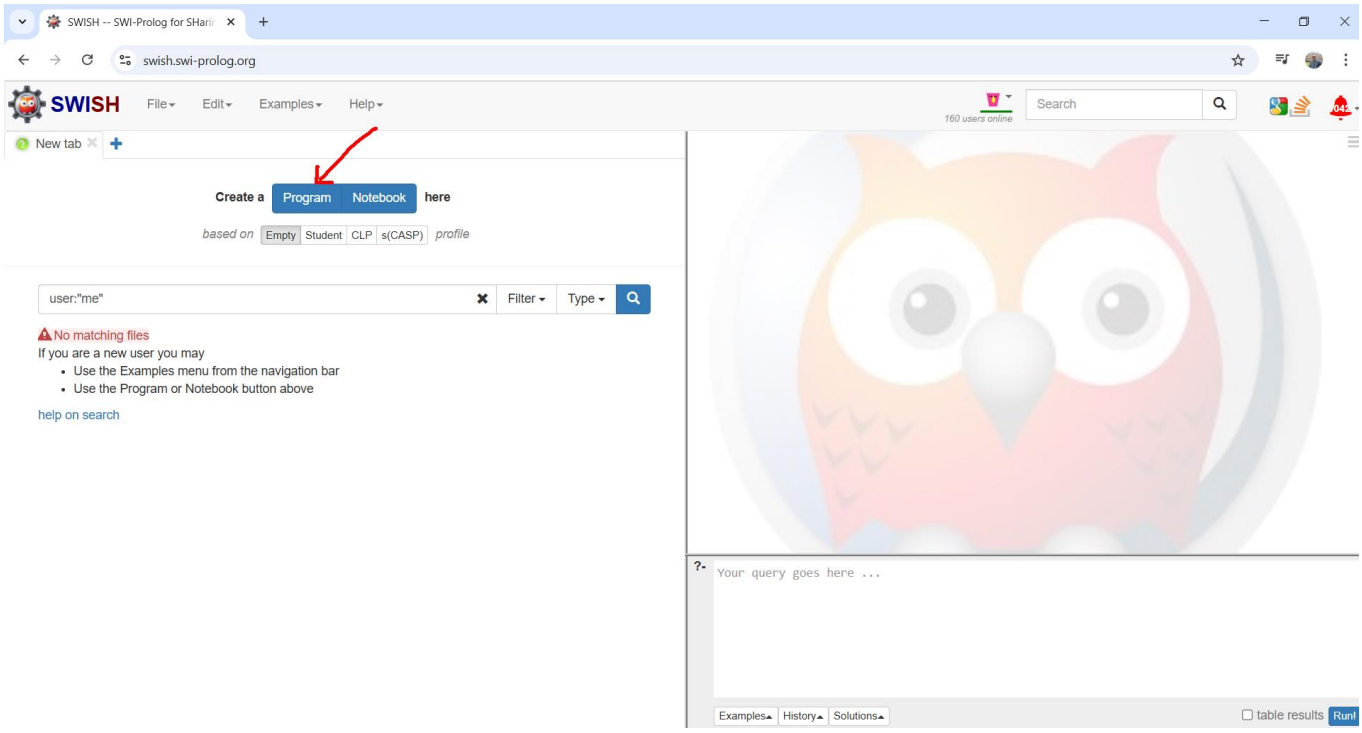
"Who is the father of Mary? Store the answer in variable X."

Program1: Demo on Facts, Rules, Variables and Queries

Step1. Go to

https://swish.swi-prolog.org/

Step1. Create a new Program

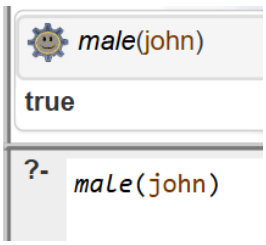


Step3. Write Prolog language instructions / code

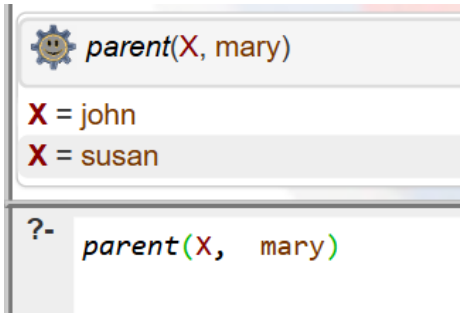
```
% FACTS

parent(john, mary).
parent(susan, mary).
parent(john, tom).
parent(susan, tom).
male(john).
male(tom).
female(susan).
female(mary).

% RULES

father(Father, Child) :- parent(Father, Child), male(Father).
mother(Mother, Child) :- parent(Mother, Child), female(Mother).
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.
```

Step4. Run the query

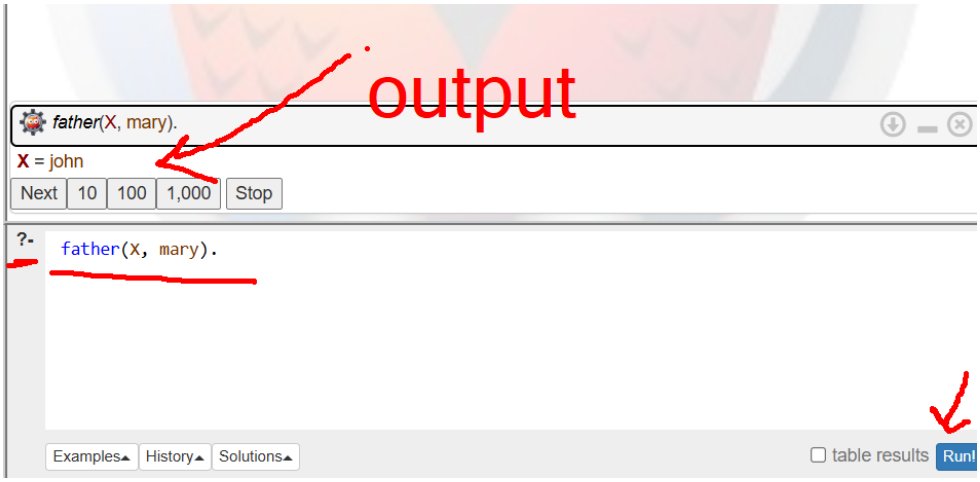
Query1:



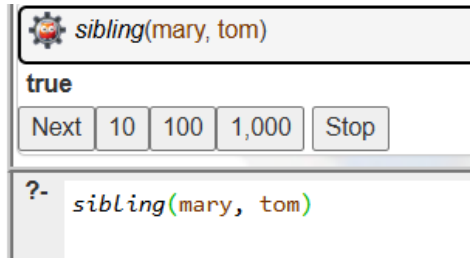
Query2:

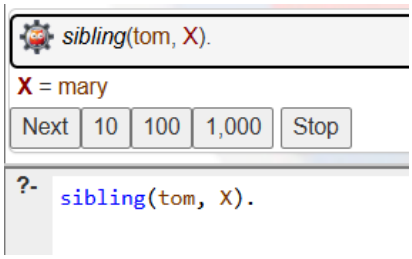


Query3:

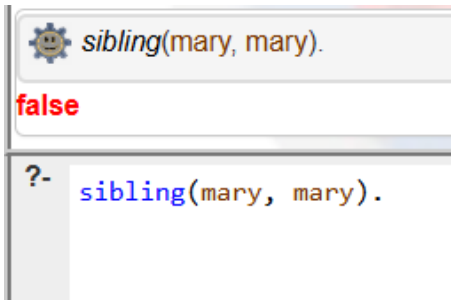


Query4:





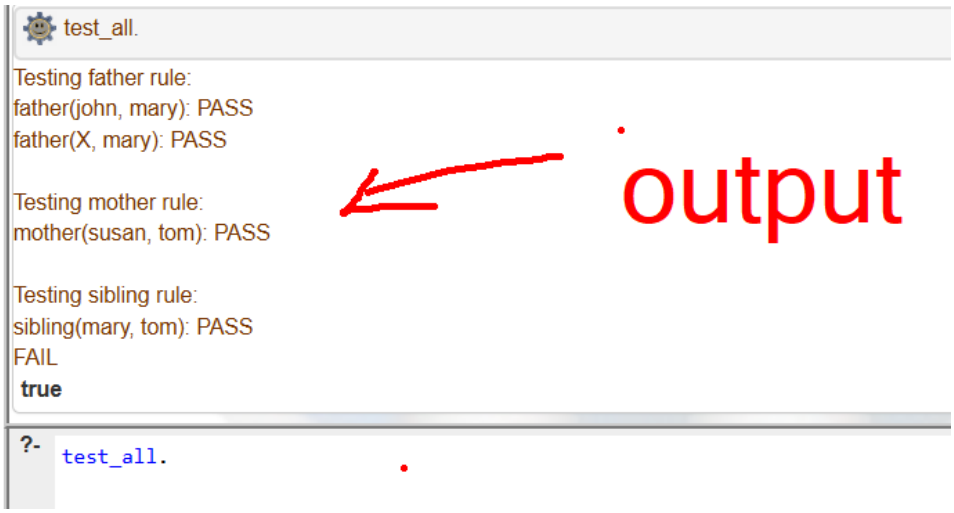
Query5:



Note: We can also create Testing Predicate (in the code area itself):

```
% You can also create a testing predicate
test_all :-
    write('Testing father rule:'), nl,
    (father(john, mary) -> write('father(john, mary): PASS') ; write('FAIL')), nl,
    (father(X, mary), X == john -> write('father(X, mary): PASS') ; write('FAIL')), nl, nl,

    write('Testing mother rule:'), nl,
    (mother(susan, tom) -> write('mother(susan, tom): PASS') ; write('FAIL')), nl, nl,

    write('Testing sibling rule:'), nl,
    (sibling(mary, tom) -> write('sibling(mary, tom): PASS') ; write('FAIL')), nl,
    (sibling(mary, mary) -> write('sibling(mary, mary): PASS') ; write('FAIL')), nl.
```

Then in the Query window:



How Prolog processes it:

1. **Pattern matching**: Prolog looks for father/2 facts/rules where the second argument is mary
2. **Unification**: Prolog tries to make father(X, mary) match with existing facts/rules
3. **Binding**: When it finds father(john, mary), it binds X = john
4. **Result**: Returns X = john as the answer

What is father/2 ?

father/2 is called the **functor** or **predicate indicator**, where:

- father is the predicate name
- /2 means it takes 2 arguments

```
% These are TWO DIFFERENT predicates:
father(X, Y).     % father/2 - relationship between two people
father(X).        % father/1 - property of one person (is a father)

% Example usage:
father(john, mary).  % Uses father/2
father(john).        % Uses father/1 (if this fact exists)
```

Types of Variables

- **Named Variables** (e.g., X, Y): Used when you want to track or reuse the value in a query or rule.
- **Anonymous Variable (_)**: Used when the value doesn't matter and won't be reused.

Program: Example AI Application in Prolog – A simple diagnostic tool for medical conditions

Requirement: In this application, the Prolog program would define a set of rules and facts about different medical conditions and their symptoms, and then use those rules and facts to diagnose a patient's condition based on their reported symptoms.

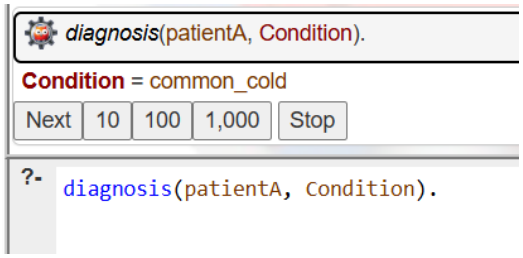
Step1.

```
% --- Facts: Patient Symptoms ---
% These facts represent the symptoms a patient is currently experiencing.
% The format is: symptom(PatientName, Symptom).

symptom(patientA, runny_nose).
symptom(patientA, sore_throat).
symptom(patientA, sneezing).
symptom(patientA, headache).

symptom(patientB, high_fever).
symptom(patientB, body_aches).
symptom(patientB, chills).
symptom(patientB, headache).

symptom(patientC, high_fever).
symptom(patientC, nausea).
symptom(patientC, vomiting).
symptom(patientC, diarrhea).
```

```
% --- Rules: Medical Conditions ---
% These rules define what constitutes a medical condition based on the presence
% of specific symptoms. The format is: diagnosis(Patient, Condition) :- Symptoms.

% Rule for a common cold
diagnosis(Patient, common_cold) :-
    symptom(Patient, runny_nose),
    symptom(Patient, sore_throat),
    symptom(Patient, sneezing).

% Rule for the flu
diagnosis(Patient, flu) :-
    symptom(Patient, high_fever),
    symptom(Patient, body_aches),
    symptom(Patient, chills).

% Rule for food poisoning
diagnosis(Patient, food_poisoning) :-
    symptom(Patient, nausea),
    symptom(Patient, vomiting),
    symptom(Patient, diarrhea).
```

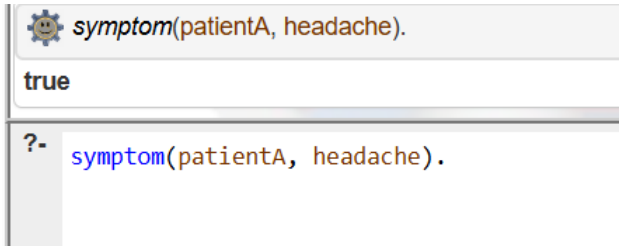
Step2. Execute some queries

Query1:



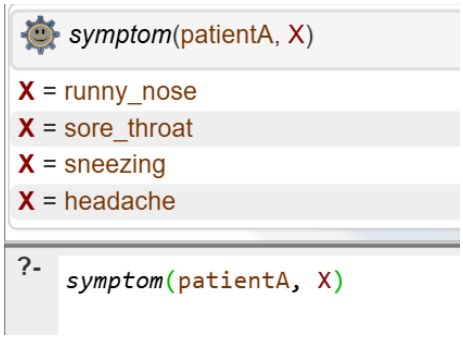
Note: Observe that: “patientA” is a Constant and “Condition” is a variable

Query2:



Note: Observe that, here both are constants.

Query3:



Program: Prolog program for a simple financial application

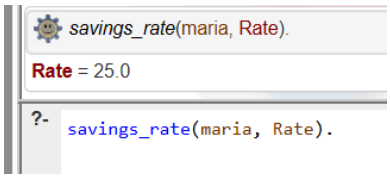
Requirement: This program will act as a basic financial advisor, using rules to analyse an individual's financial health based on income, savings, and debt.

```
% --- Facts: Individual Financial Data ---
% This section contains facts about individuals.
% The format is: financial_data(Name, MonthlyGrossIncome, MonthlySavings, MonthlyDebt).

financial_data(maria, 6000, 1200, 500).
financial_data(david, 4500, 300, 1000).
financial_data(sara, 8000, 3000, 800).
financial_data(john, 3500, 50, 200).
financial_data(emily, 9000, 4000, 1500).
```

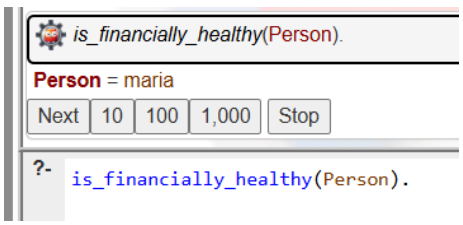
```
% --- Rules: Financial Calculations and Assessments ---
% These rules calculate various financial metrics.
% Rule to calculate net income (assuming a flat 20% tax rate).
net_income(Person, Net) :-
    financial_data(Person, Gross, _, _),
    Net is Gross * 0.80.

% Rule to calculate the savings rate as a percentage of net income.
% The savings rate is a key indicator of financial health.
savings_rate(Person, Rate) :-
    net_income(Person, Net),
    financial_data(Person, _, Savings, _),
    Rate is (Savings / Net) * 100.

% Rule to calculate the debt-to-income ratio.
% A lower ratio is generally better.
debt_to_income_ratio(Person, Ratio) :-
    net_income(Person, Net),
    financial_data(Person, _, _, Debt),
    Ratio is (Debt / Net) * 100.

% Rule for a high-level financial health assessment.
% This combines multiple criteria to provide a single diagnosis.
is_financially_healthy(Person) :-
    savings_rate(Person, Rate),
    debt_to_income_ratio(Person, Ratio),
    % A person is considered financially healthy if their savings rate is
    % above 15% AND their debt-to-income ratio is below 30%.
    Rate > 15,
    Ratio < 30.

```

Step2.

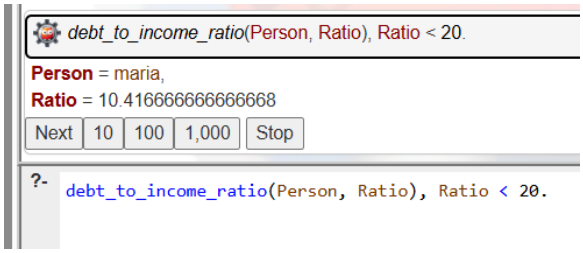
Query1:



Query2:



Query3:



Lists in Prolog

It is a fundamental data structure for representing collections of items.
a Prolog list is a data structure that can hold any number of elements.
Unlike arrays in other languages, Prolog lists are defined recursively using two parts:

- The **Head**: The first element of the list.
- The **Tail**: The rest of the list, which is itself another list.

This simple [Head | Tail] structure is incredibly powerful because it allows you to easily process a list one element at a time, which is perfect for recursive programming.

Prolog Lists: A Recursive Data Structure – search for a member

Let's break down what happens with a step-by-step example. Say you run the query `?- member(banana, [apple, banana, cherry]).`

1. **Prolog tries the first rule:** `member(Element, [Element|_]).`
   - It tries to match `banana` with the `Head` of the list `[apple, banana, cherry]`.
   - The head is `apple`, which does not match `banana`. So, this rule **fails**. Prolog doesn't stop here; it moves on to the next rule for the same predicate.

2. **Prolog tries the second rule:** `member(Element, [_|Tail]) :- member(Element, Tail).`
   - The `[_|Tail]` pattern matches the list `[apple, banana, cherry]`.
   - The `_` (anonymous variable) unifies with `apple`, but Prolog doesn't care about it.
   - The `Tail` variable unifies with the rest of the list, which is `[banana, cherry]`.
   - The rule says, "To prove that `banana` is a member of `[apple, banana, cherry]`, you must prove that `banana` is a member of the `Tail`, which is `[banana, cherry]`."

3. **The Recursive Call:** Prolog now starts the process all over again with the new, shorter list: `member(banana, [banana, cherry]).`
   - It tries the first rule again.
   - This time, the `Head` of the list is `banana`, which **matches** the `Element` we are looking for!
   - The rule succeeds, and because all the steps in the chain succeeded, the original query `?- member(banana, [apple, banana, cherry]).` also succeeds.

This is the essence of recursion in Prolog. The first rule is the **base case** (the "stop" condition when a match is found), and the second rule is the **recursive case** (the "keep going" condition that tries again with a smaller problem). The two rules work together to check every element in the list until a match is found or the list runs out of elements.

Prolog Lists: A Recursive Data Structure – find the length of a list

The Goal: Find the length of the list `[a, b, c, d, e]`.

Initial Query: `?- list_length([a, b, c, d, e], Length).`

Step 1: Prolog looks at the first rule: `list_length([], 0).`

- It tries to match `[a, b, c, d, e]` with `[]`. They do not match.
- The first rule fails, so Prolog moves on to the next one.

Step 2: Prolog looks at the second rule: `list_length([_|Tail], Length) :- ...`

- It successfully matches `[a, b, c, d, e]` with the pattern `[_|Tail]`.
- The `_` matches `a` (which is ignored).
- The `Tail` variable is unified with the rest of the list: `[b, c, d, e]`.
- The `Length` variable is still unbound.
- Prolog now has a new, sub-goal to solve: `list_length([b, c, d, e], TailLength)`.

Step 3: Now Prolog works on the new sub-goal: `list_length([b, c, d, e], TailLength)`.

- It tries the first rule again, which fails.
- It tries the second rule again, matching `[b, c, d, e]` with `[_|Tail]`.
- The `_` matches `b`.
- The `Tail` is now `[c, d, e]`.
- It creates a new sub-goal: `list_length([c, d, e], TailLength)`.

Step 4: This process repeats.

- **Query:** `list_length([c, d, e], TailLength)`
- **Tail:** `[d, e]`
- **New Sub-Goal:** `list_length([d, e], TailLength)`

Step 5:

- **Query:** `list_length([d, e], TailLength)`
- **Tail:** `[e]`
- **New Sub-Goal:** `list_length([e], TailLength)`

Step 6:

- **Query:** `list_length([e], TailLength)`
- **Tail:** `[]`
- **New Sub-Goal:** `list_length([], TailLength)`

Step 7: The Base Case is Hit!

- **Query:** `list_length([], TailLength)`
- Prolog tries the first rule again: `list_length([], 0).`
- The pattern matches perfectly!
- Prolog successfully unifies `TailLength` with `0`.
- The current sub-goal is now solved.

Step 8: "Winding Up" the Recursion

Now that the final sub-goal is solved, Prolog goes back up the chain of calls, applying the last line of the recursive rule: `Length is TailLength + 1`.

- **From Step 6:** `Length is 0 + 1`. `Length` becomes `1`.
- **From Step 5:** `Length is 1 + 1`. `Length` becomes `2`.
- **From Step 4:** `Length is 2 + 1`. `Length` becomes `3`.
- **From Step 3:** `Length is 3 + 1`. `Length` becomes `4`.
- **From Step 2:** `Length is 4 + 1`. `Length` becomes `5`.

Final Result: Prolog successfully unifies the original Length variable with 5 and returns the final answer. This "unwinding" process is characteristic of Prolog's backward-chaining inference. It finds a path to a successful base case and then works its way back up, binding variables as it goes.

Prolog Lists: A Recursive Data Structure – append a list

Base Case: `list_append([], L, L).`

This is the simplest and most fundamental rule. It provides the "stopping condition" for the recursion. It can be read as a simple fact:

> "The result of appending an empty list (`[]`) to any list `L` is just `L` itself."

This is the non-recursive case, and it's essential for terminating the process. Without it, the recursive calls would never end.

Recursive Case: `list_append([H|T1], L2, [H|T3]) :- list_append(T1, L2, T3).`

This rule defines the logic for how to handle non-empty lists. It's the "keep going" part of the process. This rule says:

> "To append a list that has a `Head` and a `Tail` (`[H|T1]`) to another list (`L2`), the resulting list will have that same `Head` (`H`) followed by a new list (`T3`). This new list `T3` is what you get when you recursively append the `Tail` (`T1`) to the original second list (`L2`)."

This rule works by peeling off the first element (`H`) from the first list and holding onto it. It then hands off the rest of the problem—appending the rest of the first list (`T1`) to the second list (`L2`)—to a new, recursive call of `list_append` .

A Step-by-Step Trace

Let's trace a simple query: `?- list_append([a, b], [c, d], Result).`

1. **Initial Call:** `list_append([a, b], [c, d], Result)`
    - The first rule fails because the first list is not empty.
    - The second rule matches.
    - `H` unifies with `a` .
    - `T1` unifies with `[b]` .
    - `L2` unifies with `[c, d]` .
    - `Result` is now unified with the pattern `[a|T3]` , but `T3` is still a variable.
    - The new recursive goal is `list_append([b], [c, d], T3)` .

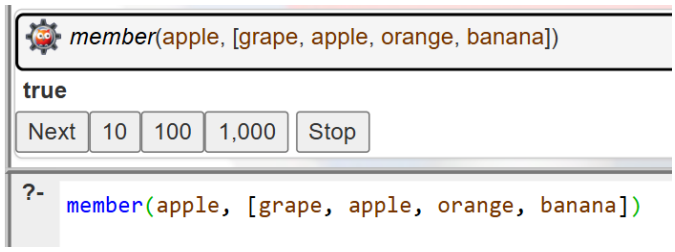
2. **Second Recursive Call:** `list_append([b], [c, d], T3)`
    - The first rule fails.
    - The second rule matches.
    - `H` unifies with `b` .
    - `T1` unifies with `[]` .
    - `L2` unifies with `[c, d]` .
    - `T3` is now unified with the pattern `[b|T4]` , but `T4` is a new variable.
    - The new recursive goal is `list_append([], [c, d], T4)` .

3. **Base Case:** `list_append([], [c, d], T4)`
   - The first rule **succeeds**! It matches `[]` with `[]` and `L` with `[c, d]`.
   - This unifies `T4` with `[c, d]`.
   - This goal is now complete.
4. **Winding Up the Recursion:** Now Prolog goes back up the call stack, binding variables as it goes.
   - **From Call 2:** `T3` was `[b|T4]`. Since `T4` is now `[c, d]`, `T3` becomes `[b|[c,d]]`, which simplifies to `[b, c, d]`.
   - **From Call 1:** `Result` was `[a|T3]`. Since `T3` is now `[b, c, d]`, `Result` becomes `[a|[b,c,d]]`, which simplifies to `[a, b, c, d]`.
5. **Final Result:** The query succeeds, and `Result` is bound to `[a, b, c, d]`.

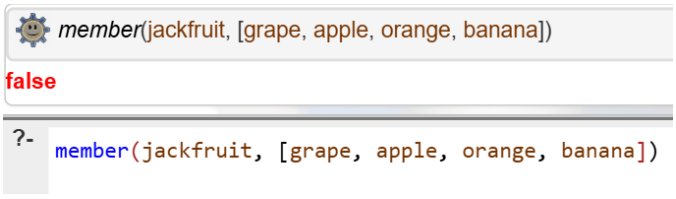
Program: Demo on Prolog Lists – check if an element is a member of a list

```
% --- Predicate: member/2 ---
% This predicate checks if an element is a member of a list.
% It's a great example of recursion and pattern matching.

% Base Case: The element is the Head of the list.
member(Element, [Element|_]).

% Recursive Case: The element is in the Tail of the list.
member(Element, [_|Tail]) :-
    member(Element, Tail).
```

Query1:



Query2:



Program: Demo on Prolog Lists – find length of a list

```prolog
% --- Predicate: length/2 ---
% This predicate calculates the number of elements in a list.

% Base Case: The length of an empty list is 0.
list_length([], 0).

% Recursive Case: The length of a list is 1 plus the length of its Tail.
list_length([_|Tail], Length) :-
    list_length(Tail, TailLength),
    Length is TailLength + 1.

```

Query1:

```
list_length([], Length).
Length = 0
```

```
?- list_length([], Length).
```

Query2:

```
list_length([a, b, c, d, e], Length).
Length = 5
```

```
?- list_length([a, b, c, d, e], Length).
```

Program: Demo on Prolog Lists – append a list

```prolog
% --- Predicate: append/3 ---
% This predicate joins two lists together. It's a powerful tool.

% Base Case: Appending an empty list to a list L gives you L.
list_append([], L, L).

% Recursive Case: To append a list L2 to [H|T1], the result is [H|T3],
% where T3 is the result of appending L2 to T1.
list_append([H|T1], L2, [H|T3]) :-
    list_append(T1, L2, T3).

```

Query:

```
list_append([a, b], [c, d], Result).
Result = [a, b, c, d]
```

```
?- list_append([a, b], [c, d], Result).
```

Predicate logic, Unification and Backward Reasoning

- **Predicate Logic**: How you express facts and rules
- **Unification**: How Prolog matches patterns and binds variables

- **Backward Reasoning/ Backward Chaining**: How Prolog solves queries by working backward from goals.

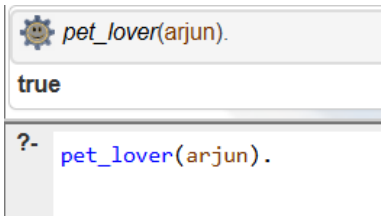
  It is Prolog's default method for solving queries. It starts with the query (goal) and works backward to find facts or rules that satisfy it. Prolog tries to prove the goal by matching it to facts or reducing it to sub-goals via rules.

Program: Demo on Predicate Logic

Note: A pet ownership scenario

```
% Facts
owns(arjun, cat).
owns(kalyan, dog).

% Rule: If someone owns a pet, they are a pet lover
pet_lover(X) :- owns(X, _).
% Here, _ means "there exists some value for the second argument of owns, but I don't care what it is."

```

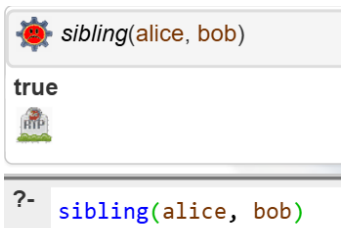
Query:



Program: Demo on "Unification"

```
% Fact: States that John is the parent of Alice
parent(john, alice).

% Fact: States that Mary is the parent of Alice
parent(mary, alice).

% Fact: States that John is the parent of Bob
parent(john, bob).

% Rule: Defines that X and Y are siblings if they share the same parent Z and are not the same person
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
%This is a rule, not a fact.
%the LHS part is Rule's Head and the RHS part is Rule's Body

```

Query:



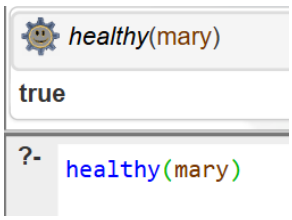
Unification process:

- **Unification Process**:
  - Unify `sibling(alice, bob)` with the rule's head `sibling(X, Y)`, binding `X = alice`, `Y = bob`.
  - Check the rule's body: `parent(Z, alice)` and `parent(Z, bob)`.
  - For `parent(Z, alice)`: Unifies with `parent(john, alice)` (binding `Z = john`) or `parent(mary, alice)` (binding `Z = mary`).
  - For `parent(Z, bob)`: Unifies with `parent(john, bob)` (binding `Z = john`).
  - Since `Z = john` satisfies both `parent(john, alice)` and `parent(john, bob)`, and `alice \= bob` is true, the query succeeds.

Program: Demo on Backward chaining

```
% Facts
eats(john, pizza).
eats(mary, salad).

% Rule: If someone eats healthy food, they are healthy
healthy(X) :- eats(X, salad).

```

Query:



Backward chaining process:

Backward Reasoning Process:

- Query: `healthy(mary)`.
- Prolog looks for a fact or rule matching `healthy(mary)`.
- Finds the rule `healthy(X) :- eats(X, salad)`.
- Unifies `X = mary`, reducing the goal to `eats(mary, salad)`.
- Checks facts: `eats(mary, salad)` is true.
- Therefore, `healthy(mary)` is true.

Output: `true`.

Unification Vs Backward Chaining

- **Unification** is a **low-level mechanism** that happens as part of Prolog's evaluation process. It's about matching terms (e.g., sibling(alice, bob) with sibling(X, Y)) by assigning values to variables.
- **Backward Chaining** is a **higher-level strategy** that determines the order in which Prolog searches for solutions. It uses unification to match queries with rules or facts and to resolve sub-goals, but it encompasses the entire process of working backward from the query to the knowledge base.

Matching Techniques and Matching Variables

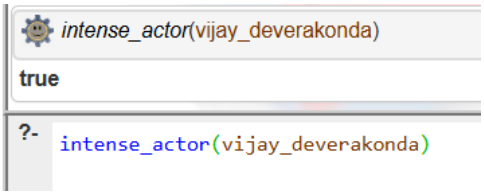
Matching Techniques: Refer to unification methods, such as **exact matching** (constants match exactly), **variable binding** (variables bind to values), and **anonymous variable matching** (using _ for "don't care" terms).

Matching Variables: Variables (e.g., X, Y, _) are placeholders that unify with values during matching. Named variables track values, while _ ignores them.

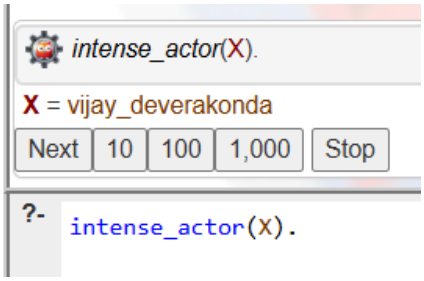
Program: Demo on Matching Techniques and Matching Variables

```
% Facts: Actor acted in a movie
acted(vijay_deverakonda, arjun_reddy).  % Vijay Deverakonda acted in Arjun Reddy
acted(prabhas, baahubali).              % Prabhas acted in Baahubali

% Facts: Movie is intense
intense(arjun_reddy).                   % Arjun Reddy is an intense movie

% Rule: An actor is intense if they acted in an intense movie
intense_actor(X) :- acted(X, Y), intense(Y).
```

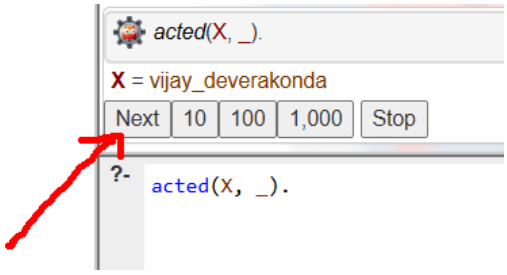
Query1: Is Vijay Deverakonda an intense actor?



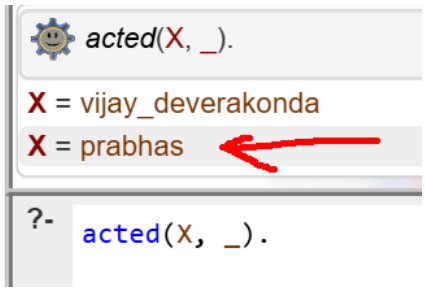
Query2: Who is an intense actor?



Query3: Does anyone act in any movie?



Click “Next” to view the next value in the output



Matching Techniques:

- **Exact Matching (Query 1)**:
  - Query: `?- intense_actor(vijay_deverakonda).`
  - Unifies `intense_actor(vijay_deverakonda)` with `intense_actor(X)`, binding `X = vijay_deverakonda`.
  - Sub-goal `acted(vijay_deverakonda, Y)` unifies with `acted(vijay_deverakonda, arjun_reddy)`, binding `Y = arjun_reddy`.
  - Sub-goal `intense(arjun_reddy)` matches the fact directly.
  - **Output**: `true`.
- **Variable Binding (Query 2)**:
  - Query: `?- intense_actor(X).`
  - Unifies `X` with `vijay_deverakonda` (via `acted(vijay_deverakonda, arjun_reddy)` and `intense(arjun_reddy)`).
  - **Output**: `X = vijay_deverakonda`.
- **Anonymous Variable Matching (Query 3)**:
  - Query: `?- acted(X, _).`
  - The `_` matches any movie (e.g., `arjun_reddy` or `baahubali`) without binding it.
  - Unifies `X` with `vijay_deverakonda` and `prabhas`.
  - **Output**: `X = vijay_deverakonda` (press `;` for `X = prabhas`).

Matching Variables:

- `X`: Used to track actors (e.g., binds to `vijay_deverakonda` or `prabhas`).
- `Y`: Used to track movies in rules (e.g., binds to `arjun_reddy`).
- `_`: Ignores the movie name when it’s not needed (e.g., in `acted(X, _)`).

Backward Reasoning Vs Forward Reasoning

Backward Reasoning (Goal-Driven): This is the primary mode of operation for Prolog. The system starts with a **goal** (your query) and **works backward**, trying to find facts that support that goal. It asks, "To prove this, what must be true?" and then finds the sub-goals to prove. It's like asking, "Did the suspect commit the crime?" and then looking for evidence (facts) to support or refute that specific hypothesis.

Forward Reasoning (Data-Driven): This is the opposite. The system starts with a set of **facts** and reasons forward, applying all possible rules to derive new conclusions until no new conclusions can be made. It asks, "**Given this new information, what new things do I now know to be true?**" It's like a detective observing a piece of evidence and then considering all the possible crimes that piece of evidence could point to. Forward reasoning is common in **expert systems** for situations like medical monitoring, where new data arrives continuously, and the system needs to react.

Ex: A Patient Monitoring System

Imagine a patient in the ICU. They are connected to various sensors that continuously send data to a central computer system. This system's job is to watch for dangerous health changes in real time.

Key Components:

- **Facts (The Data):** This is the raw information flowing into the system.
    - Heart rate: 85 bpm
    - Blood pressure: 120/80
    - Oxygen saturation: 98%
    - Temperature: 37°C
- **Rules (The Medical Knowledge):** These are the pre-programmed "if-then" statements that define logical relationships. The system knows these rules are true.
    - **Rule A: IF** heart rate is above 120 bpm **AND** blood pressure is below 90/60, **THEN** issue a "Hypovolemic Shock" alert.
    - **Rule B: IF** oxygen saturation is below 90%, **THEN** issue an "Oxygen Alert."
    - **Rule C: IF** temperature is above 39°C, **THEN** issue a "Fever Alert."

The Step-by-Step Reasoning Process:

The system operates in a continuous loop, always looking for a match between the incoming facts and its set of rules.

Step 1: Initial State The system is running. All the patient's vitals are stable and within normal ranges. The system has a list of facts (current vitals) and a list of rules. Nothing is happening because no rule's "IF" conditions are met.

Step 2: A New Fact Arrives (The "Trigger") The patient's heart rate starts to climb. The sensor sends a new data point to the system.

- **New Fact:** Heart rate is now 125 bpm.

Step 3: The System Checks for a Match The system's reasoning engine receives this new fact and checks it against all of its rules.

- Does it match Rule A? The first part of the rule (heart rate above 120 bpm) is now true. However, the second part (blood pressure below 90/60) is still false. So, Rule A's conditions are not fully met yet.
- Does it match Rule B? No, because the new fact is about heart rate, not oxygen saturation.
- Does it match Rule C? No, because the new fact is about heart rate, not temperature.

- The system moves on, continuing to monitor.

Step 4: Another Key Fact Arrives The patient's blood pressure begins to drop. The sensor sends a new data point.

- **New Fact:** Blood pressure is now 70/45.

Step 5: A Full Match is Found, and a Conclusion is Reached The system again checks all its rules against the new and existing facts.

- It looks at **Rule A** again. The "IF" conditions are now both met:
    1. The fact "heart rate is 125 bpm" is true.
    2. The new fact "blood pressure is 70/45" is true.
- Because all the conditions are met, the system "fires" the rule.

Step 6: The Action is Taken The "THEN" part of the rule is executed.

- The system immediately issues a "Hypovolemic Shock" alert, sounding an alarm and notifying a nurse. This new alert is also added as a fact to the system's knowledge base.

Combining Them: In more complex systems, a hybrid approach is often used. For instance, a system might use forward reasoning to process new data and generate a list of potential hypotheses (goals), and then use backward reasoning to explore a specific hypothesis in detail. **Prolog itself can be made to simulate forward reasoning, but its native and most efficient method is backward reasoning.**

In the hospital, forward reasoning is perfect for continuous monitoring and triggering alerts, while a doctor might use backward reasoning to diagnose a condition after the alert has been triggered.

Program: Demo on Backward and Forward Reasoning - Medical Advisor Application

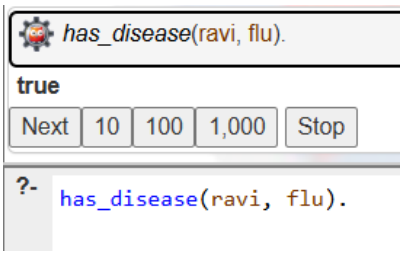
Requirement:

Backward reasoning mimics a doctor asking, “Does this patient have flu?” while **Forward** reasoning mimics a doctor listing all possible diagnoses based on symptoms.

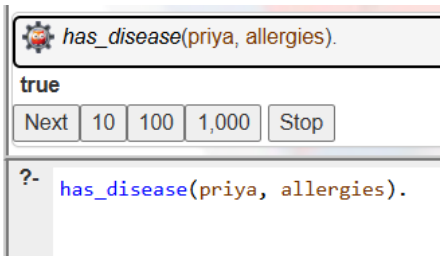
```
% Fact: Ravi has a fever
has_symptom(ravi, fever).        % Declares that patient Ravi exhibits the symptom of fever

% Fact: Ravi has a cough
has_symptom(ravi, cough).        % Declares that patient Ravi exhibits the symptom of cough

% Fact: Ravi has a sore throat
has_symptom(ravi, sore_throat).  % Declares that patient Ravi exhibits the symptom of sore throat

% Fact: Ravi has fatigue
has_symptom(ravi, fatigue).      % Declares that patient Ravi exhibits the symptom of fatigue

% Fact: Priya has a cough
has_symptom(priya, cough).       % Declares that patient Priya exhibits the symptom of cough

% Fact: Priya has a headache
has_symptom(priya, headache).    % Declares that patient Priya exhibits the symptom of headache

% Rule: A patient has flu if they have fever and cough
has_disease(X, flu) :- has_symptom(X, fever), has_symptom(X, cough).

% Rule: A patient has cold if they have cough and sore throat
has_disease(X, cold) :- has_symptom(X, cough), has_symptom(X, sore_throat).

% Rule: A patient has bronchitis if they have cough and fatigue
has_disease(X, bronchitis) :- has_symptom(X, cough), has_symptom(X, fatigue).

% Rule: A patient has allergies if they have headache and cough
has_disease(X, allergies) :- has_symptom(X, headache), has_symptom(X, cough).
```

Output:

Query1: Query for backward reasoning: Does Ravi have the flu?



Query2: Query for backward reasoning: Does Priya have allergies?



Query3: Query for forward reasoning: Find all diseases Ravi might have

```
findall(Disease, has_disease(ravi, Disease), RaviDiseases).
RaviDiseases = [flu, cold, bronchitis]
```

```
?- findall(Disease, has_disease(ravi, Disease), RaviDiseases).
```

Query4: Find all diseases Priya might have

```
findall(Disease, has_disease(priya, Disease), PriyaDiseases)
PriyaDiseases = [allergies]
```

```
?- findall(Disease, has_disease(priya, Disease), PriyaDiseases)
```

The query `?- findall(Disease, has_disease(ravi, Disease), RaviDiseases).` is designed to find all diseases Ravi might have based on these facts and rules.

Explanation:

Let's walk through how Prolog processes this query:

1. **Initialization**:
   - Prolog starts with the query `?- findall(Disease, has_disease(ravi, Disease), RaviDiseases).`.
   - It initializes an empty list for `RaviDiseases` and prepares to collect values for `Disease`.
2. **Evaluating the Goal**:
   - The goal is `has_disease(ravi, Disease)`, which means Prolog will try to find all values of `Disease` for which `has_disease(ravi, Disease)` is true.
   - Prolog looks at the rules defining `has_disease/2` and matches them against Ravi's symptoms.
3. **Applying the Rules**:
   - **Rule 1:** `has_disease(X, flu) :- has_symptom(X, fever), has_symptom(X, cough).`
     - Unify `X = ravi`.
     - Check sub-goals:
       - `has_symptom(ravi, fever)` → True (fact exists).
       - `has_symptom(ravi, cough)` → True (fact exists).
     - Since both sub-goals succeed, `has_disease(ravi, flu)` is true, so add `flu` to the list.

- **Rule 2:** `has_disease(X, cold) :- has_symptom(X, cough), has_symptom(X, sore_throat).`
  - Unify `X = ravi`.
  - Check sub-goals:
    - `has_symptom(ravi, cough)` → True.
    - `has_symptom(ravi, sore_throat)` → True.
  - Since both sub-goals succeed, `has_disease(ravi, cold)` is true, so add `cold` to the list.
- **Rule 3:** `has_disease(X, bronchitis) :- has_symptom(X, cough), has_symptom(X, fatigue).`
  - Unify `X = ravi`.
  - Check sub-goals:
    - `has_symptom(ravi, cough)` → True.
    - `has_symptom(ravi, fatigue)` → True.
  - Since both sub-goals succeed, `has_disease(ravi, bronchitis)` is true, so add `bronchitis` to the list.

- **Rule 4:** `has_disease(X, allergies) :- has_symptom(X, headache), has_symptom(X, cough).`
  - Unify `X = ravi`.
  - Check sub-goals:
    - `has_symptom(ravi, headache)` → False (no such fact).
    - Fails, so `has_disease(ravi, allergies)` is not true, and `allergies` is not added.

4. **Collecting Results**:
   - After evaluating all rules, the values of `Disease` that worked are `flu`, `cold`, and `bronchitis`.
   - These are collected into the list `RaviDiseases`.
5. **Output**:
   - Prolog unifies `RaviDiseases` with `[flu, cold, bronchitis]`.
   - **Final Output**: `RaviDiseases = [flu, cold, bronchitis]`.

Why This Simulates Forward Reasoning

- **Data-Driven**: Unlike backward reasoning, which starts with a specific goal (e.g., "Does Ravi have flu?"), **findall()** starts with the facts (Ravi's symptoms) and applies rules forward to derive all possible conclusions (diseases).
- **Comprehensive**: It doesn't stop at the first solution (like a single query might) but collects all diagnoses, mimicking how a doctor might consider all possibilities based on observed symptoms.

Procedural vs Declarative Knowledge

Procedural Programming

- **Definition**: Focuses on **how** a problem should be solved (step-by-step instructions).
- In languages like C, Java, or Python (when written imperatively), you explicitly tell the computer:
  - What steps to perform.
  - In what order to perform them.
- **Control is explicit**: programmer defines loops, conditionals, assignments, etc.

Declarative Programming

- **Definition**: Focuses on **what** the problem is, not how to solve it.
- In declarative style, you just specify *facts and rules* (the logic), and Prolog's inference engine figures out *how* to get the solution.
- **Control is implicit**: you don't tell Prolog "how" to solve, only "what is true".

Python Example (Procedural):

```
# Procedural function to check if student passed all subjects
def check_passed(mark1, mark2, mark3):

    if (mark1 >= 35 and mark2 >= 35 and mark3 >= 35):
        return True
    else:
        return False

# Main program to test the function
student_name = "Bhavya"
math_mark = 40
science_mark = 40
english_mark = 45

# Call the procedural function
if check_passed(math_mark, science_mark, english_mark):
    print(f"{student_name} has passed all subjects.")
else:
    print(f"{student_name} has failed in at least one subject.")
```

Output:

```
Bhavya has passed all subjects.
```

Prolog Example (declarative):

```
% Facts: Marks for each subject
mark(bhavya, math, 40).
mark(bhavya, science, 70).
mark(bhavya, english, 45).

% Rule: Student passes if all subjects have marks >= 35
passed(Student) :-
    mark(Student, math, MathMark), MathMark >= 35,
    mark(Student, science, ScienceMark), ScienceMark >= 35,
    mark(Student, english, EnglishMark), EnglishMark >= 35.
```

In this declarative Prolog example, we define facts for the student's marks and a rule that declares the student has passed if all marks are >= 35. Prolog infers the result based on the logic without specifying procedural steps.

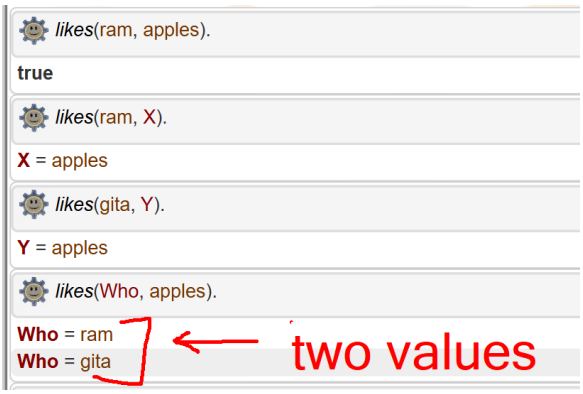
Output:

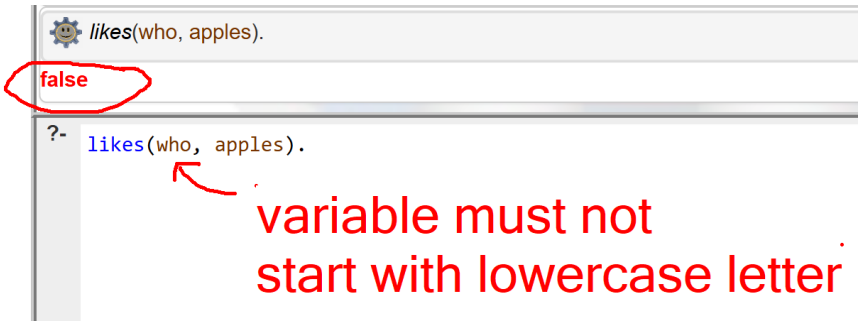
```
passed(bhavya)
true

?- passed(bhavya)
```

Program: Demo on Unification and Resolution Algorithms

Example1: Demo on Simple Unification

```
% Knowledge Base
likes(ram, apples). %ram likes apples
likes(sita, bananas).  %sita likes apples
likes(gita, X) :- likes(ram, X). %Gita likes X if Ram likes X.
%The :- operator in Prolog indicates an "if" condition or logical implication.
```



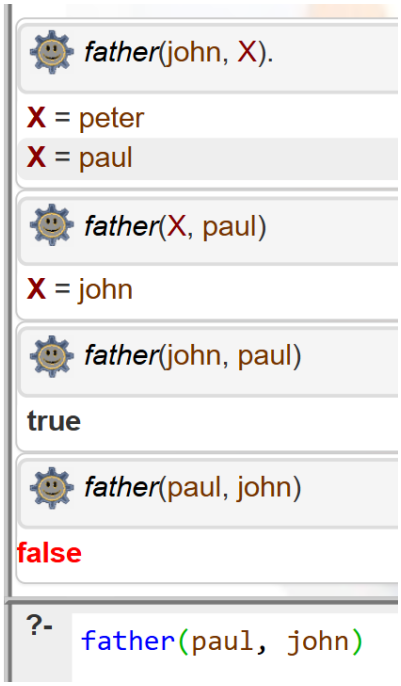


Complete Unification Flow:

1. Query: likes(gita, apples)
2. Unifies with rule head: X = apples
3. Body becomes: likes(ram, apples)
4. Unifies with fact: likes(ram, apples)
5. **Result: true**

Example2: Demo on Unification Failures

```
father(john, peter).
father(john, paul).
```



Note:

Step	Operation	Description
Matching `likes(gita, What)` with `likes(gita, X)`	**Unification**	Finds θ = { What / X }
Replacing `likes(gita, What)` with subgoal `likes(ram, X)`	**Resolution**	Applies the rule's body to continue reasoning
Matching `likes(ram, X)` with `likes(ram, apples)`	**Unification (again)**	Finds θ = { X / apples }
Using both substitutions to conclude `What = apples`	**Resolution chain result**	Derived fact

Section-3: Rule Matching and Firing in Rule-Based Systems / Rule Execution Mechanisms

RETE Algorithm (Efficient Pattern Matching)

The RETE algorithm is an **efficient pattern-matching algorithm** used in **rule-based AI systems** to **determine which rules can be fired**, without checking all rules from **scratch** every time.

Developed by Charles Forgy in the 1970s, it's a pattern-matching technique **designed to speed up this process** in production rule systems, avoiding the inefficiency of checking every rule against every fact repeatedly.

How It Works

RETE (pronounced "REE-tee," from Latin for "net/network") builds a discrimination network, which is essentially a graph or tree-like structure:

- **Alpha Nodes**: These **handle individual conditions within rules.** They filter facts based on simple tests (e.g., "is the temperature > 100 F ?"). Facts that pass are stored here to avoid retesting.
- **Beta Nodes**: These **join multiple alpha nodes** to check combined conditions (e.g., "temperature > 100 AND cough present"). They use memory to remember partial matches.
- **Terminal Nodes**: Represent **complete rule matches**. When a fact propagates through the network and reaches here, the rule is ready to "fire" (execute its action).

The key ideas are:

- **State Saving**: It ***remembers previous matches***, so only changes (new/updated/deleted facts) are propagated through the network.
- **Sharing**: Common conditions across rules are shared in the network, reducing redundancy.
- **Propagation**: Changes ripple through the network, updating matches efficiently.

This makes it much faster for large systems, trading some memory for speed.

The **RETE Algorithm** is like a memory-efficient teacher who only checks new or changed homework, instead of checking the whole class every time.

Steps of the RETE algorithm

Step 1. **Network Compile:** Rules are converted into a shared, branching network structure.
Step 2. **Fact Assertion:** New data (tokens) enter the network and get filtered (Alpha Nodes).
Step 3. **Pattern Matching:** Tokens combine with other tokens to find complex matches (Beta Nodes).
Step 4. **Rule Firing:** Complete matches go to the Conflict Set, one is chosen, and its action is executed.

Example1: Classroom Example

Imagine a teacher (AI system):

- Rules: "If a student has done Homework + brought Notebook → Give them a Star."
- Facts:
  - Raju = Homework ✅, Notebook ✅
  - Priya = Homework ✅, Notebook ❌

The RETE network splits this into two checks: **Homework?**, **Notebook?**

- Raju passes both → Rule fires → Raju gets a star ⭐

- Priya passes only one → Rule not fired

Next day: only Priya updates Notebook fact. RETE does **not** re-check Raju (saves time). It just checks Priya's update.

RETE **remembers past matches** (like "Raju already passed the Homework ✅ + Notebook ✅ test yesterday").

So when the next day comes, RETE doesn't waste time checking Raju again unless something about Raju **changes**.

Think of it like this:

- A normal algorithm = teacher checks **all students' homework every day** (even if it hasn't changed).
- RETE = teacher keeps a notebook of who already did homework yesterday.
    - Next day, teacher only checks students who brought something new or changed.

This is why RETE is **super fast** for large systems (hundreds of rules and thousands of facts).

It doesn't forget previous matches — it keeps them in a **network memory** until something changes.

In RETE:

1. **Facts are inserted, updated, or deleted** in working memory.
2. Each change creates a **token** (like a signal) that flows through the RETE network.
3. That token only moves through the **parts of the network** that are related to that fact.
4. The rest of the network (unaffected rules/facts) stays as-is.

So if "Raju's facts" don't change, no new token is generated for Raju → RETE does not re-check him.

Example2: Medical Example with Patient ID

Rules in the Knowledge Base

1. **Flu Rule**
   IF patient has Fever **AND** Cough → Diagnosis = Possible Flu
2. **Heart Rule**
   IF patient has Chest Pain **AND** High Blood Pressure → Diagnosis = Possible Heart Issue

Facts in Working Memory (Day 1)

For **Patient A (ID = 101):**

- Fact1: Patient(101, Fever) ✅
- Fact2: Patient(101, Cough) ✅
- Fact3: Patient(101, ChestPain) ❌ (not present)
- Fact4: Patient(101, HighBP) ✅

How RETE Works (Day 1)

1. RETE breaks rules into small **condition nodes** in its network:
    - Flu Rule → Fever?, Cough?

    - Heart Rule → ChestPain?, HighBP?
2. It sends each fact (Patient(101, …)) into the network.
    - Fever ✅ matches “Fever?” node.
    - Cough ✅ matches “Cough?” node.
    - ChestPain ❌ (not matched).
    - HighBP ✅ matches “HighBP?” node.
3. RETE combines matches with the **same ID**:
    - For Patient(101): Fever ✅ + Cough ✅ → Flu Rule fires.
    - For Patient(101): ChestPain ❌ + HighBP ✅ → Heart Rule not fired.

Diagnosis (Day 1): **“Patient 101 might have Flu.”**

Update on Day 2

Now new information arrives:

- Fact5: Patient(101, ChestPain) ✅ (new symptom observed).

Working Memory now contains:

- Fever ✅
- Cough ✅
- Chest Pain ✅ (new fact)
- HighBP ✅

How RETE Handles the Change (Day 2)

- RETE **does not** re-check Fever and Cough for Patient 101, because those facts already matched yesterday and are stored in memory.
- It only checks the **new/changed fact**: Chest Pain ✅.

Now RETE re-evaluates the Heart Rule:

- For Patient(101): ChestPain ✅ + HighBP ✅ → Heart Rule fires.

Diagnosis (Day 2):

- Patient 101 might have Flu (from earlier match).
- Patient 101 might have Heart Issue (from today’s new match).

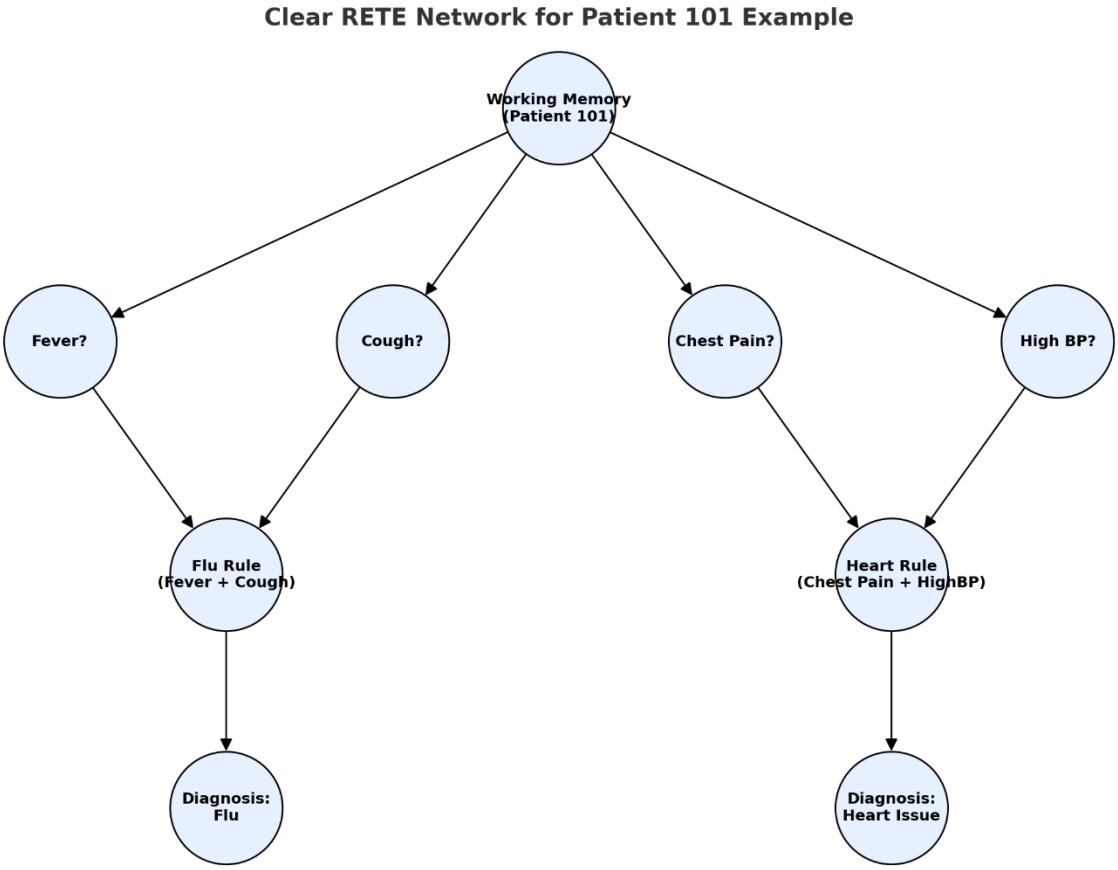
Why RETE Saves Time

Without RETE: the system would re-check **all conditions for all patients** every time.

With RETE:

- Patient(101)’s old matches (Fever + Cough → Flu) are remembered.
- On Day 2, only the **new ChestPain fact** is checked, and it triggers the Heart Rule.

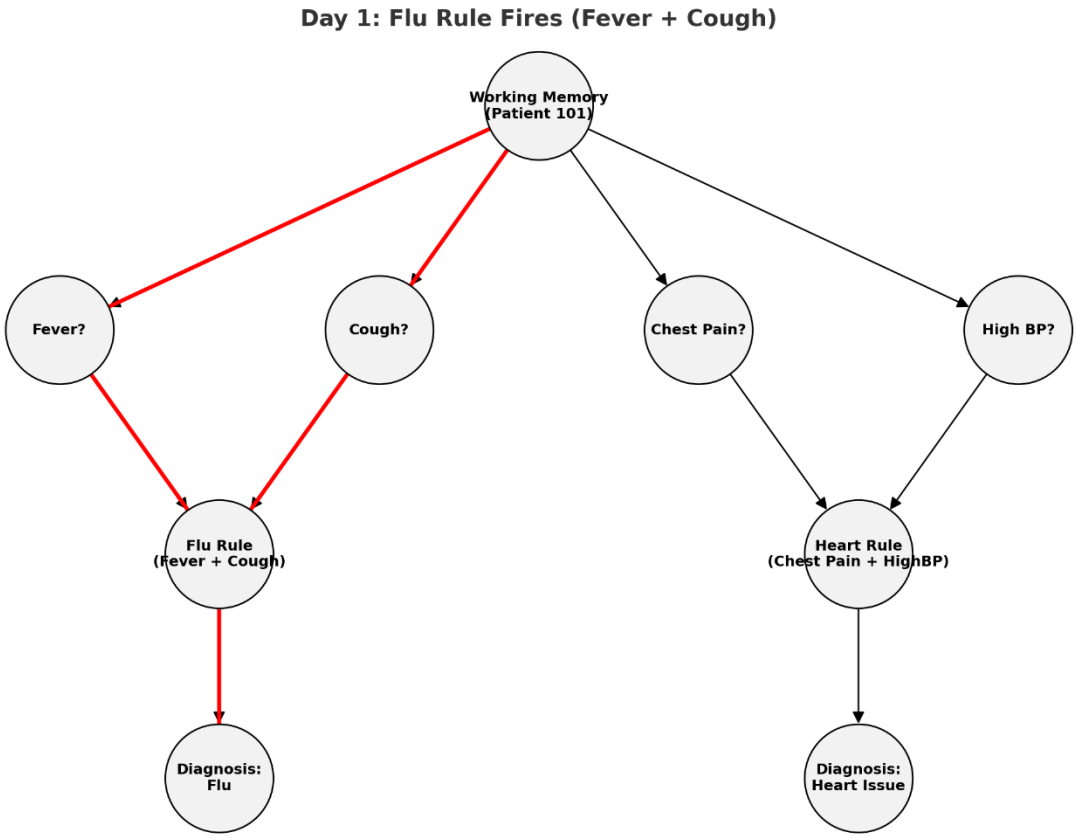
RETE Network Diagram

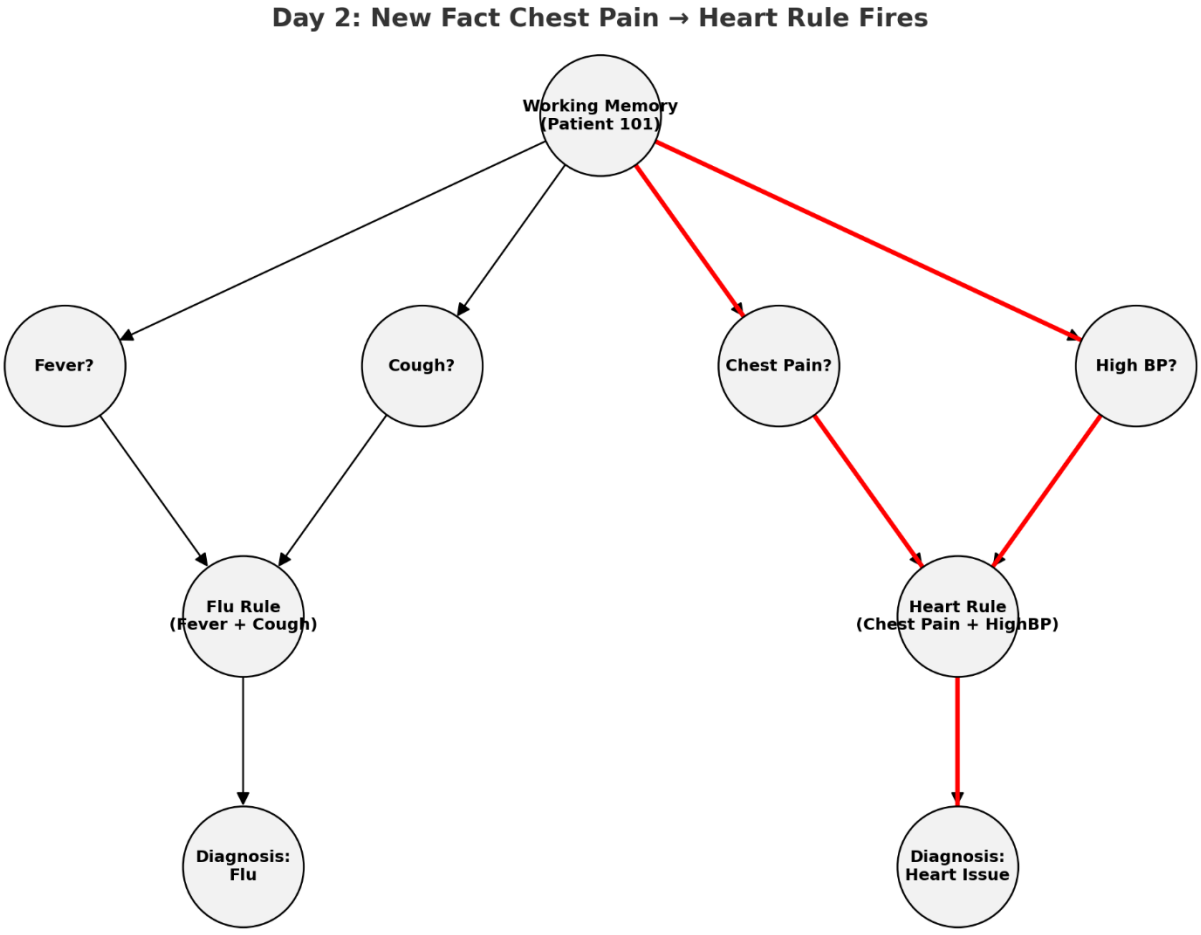


Here's the **clear layered RETE network** for Patient 101:

- **Top (Working Memory)**: Holds all facts about Patient 101.
- **Middle Layer (Conditions)**: Checks Fever?, Cough?, Chest Pain?, High BP?
- **Rule Layer**: Combines matched conditions into **Flu Rule** or **Heart Rule**.
- **Bottom Layer (Diagnosis)**: Shows which diagnosis fires.

Day wise for Patient 101 - highlighting only the *newly activated path*:





Let us see a sample implementation of RETE algorithm:

Program1: To understand a Python built-in function all()

all() checks all values → all are True → returns **True**.

Key Points About all()

1. It takes **any iterable** (list, tuple, generator, etc.).
2. Returns **True only if every element is True**.
3. Returns **False immediately** if it finds a False.
4. Returns **True for an empty iterable**.

Without using "all()" built-in function:

```
scores = [75, 82, 90, 68, 59]
passing_mark = 50

all_passed = True  # Assume all passed initially

for score in scores:
    if score < passing_mark:
        all_passed = False
        break

print("Did all students pass?", all_passed)
```

Output:

True

Using "all()" builti-in function

```
# List of students' scores
scores = [75, 82, 90, 68, 59]

# Passing mark
passing_mark = 50

# Check if all students passed
all_passed = all(score >= passing_mark for score in scores)

print("Did all students pass?", all_passed)
```

Output:

True

Program: Ex2 using all()

```
conditions = ["Fever", "Cough"]
facts1 = ["Fever", "Cough", "HighBP"]
facts2 = ["Fever", "HighBP"]

# Check if all conditions are satisfied
check1 = all(cond in facts1 for cond in conditions)
print(check1)
# True, both "Fever" and "Cough" are in facts1

check2 = all(cond in facts2 for cond in conditions)
print(check2)
# False, "Cough" is missing in facts2
```

Output:

```
True
False
>>>
```

Program: RETE implementation in Python

Note: It captures the spirit of RETE (incremental checking, working memory) but is not a full RETE network implementation.

```
# Simple RETE-like system for Medical Example

# Working Memory to store facts
# working_memory keeps a record of all facts for all patients in the form of dictionary types
working_memory = []

# Rules (each rule has conditions and a diagnosis action)
rules = [
    {
        "name": "Flu Rule",
        "conditions": ["Fever", "Cough"],
        "action": "Possible Flu"
    },
    {
        "name": "Heart Rule",
        "conditions": ["ChestPain", "HighBP"],
        "action": "Possible Heart Issue"
    }
]
```

```
# Function to assert a new fact into working memory
def assert_fact(patient_id, fact):
    entry = {"patient_id": patient_id, "fact": fact}
    #a dictionary is created to store a new fact

    #recall that "working_memory" is a list of facts/dictionaries
    #This line checks if this exact patient/fact combination already exists.
    if entry not in working_memory:
        working_memory.append(entry)
        #If the fact is not already present, it is added to working_memory.
        print(f"[INFO] Added fact: Patient {patient_id} has {fact}")
        check_rules(patient_id, fact)  # check only rules related to the new fact
```

```
def get_patient_facts(patient_id):
    return [f["fact"] for f in working_memory if f["patient_id"] == patient_id]

# Function to check rules incrementally
def check_rules(patient_id, new_fact):
    facts = get_patient_facts(patient_id)
    #get all known facts of the patient

    #Iterates through every rule in the system. Each "rule" is a dictionary
    for rule in rules:

        if new_fact in rule["conditions"]:  # Only check rules related to the new fact

            if all(cond in facts for cond in rule["conditions"]):

                #all() checks whether every condition of the rule exists in the patient's facts.
                print(f"[DIAGNOSIS] Patient {patient_id} -> {rule['action']}")


# Simulation
print("=== Day 1 ===")
assert_fact(101, "Fever")
assert_fact(101, "Cough")
assert_fact(101, "HighBP")

print("\n=== Day 2 (new info arrives) ===")
assert_fact(101, "ChestPain")  # Only Heart Rule will be checked now
```

Output:

```
=== Day 1 ===
[INFO] Added fact: Patient 101 has Fever
[INFO] Added fact: Patient 101 has Cough
[DIAGNOSIS] Patient 101 -> Possible Flu
[INFO] Added fact: Patient 101 has HighBP

=== Day 2 (new info arrives) ===
[INFO] Added fact: Patient 101 has ChestPain
[DIAGNOSIS] Patient 101 -> Possible Heart Issue
>>>
```

Note:

The **Python** example we saw **approximates RETE** by:

- Storing facts in working memory
- Incrementally checking rules based on new facts

In the **Prolog** coding we do **simple rule evaluation**, **not RETE**, because it:

- Does not store partial matches
- Evaluates rules each time from scratch when a query is made

Note: The actual RETE network Implementation is **much more complex** than the Python example and usually implemented in **expert system frameworks** like **CLIPS**, **Jess**, or **Drools**.

A full RETE implementation requires:

1. **Building alpha nodes** for each fact test.
2. **Building beta nodes** to join partial matches.
3. **Creating terminal nodes** for each rule.
4. **Maintaining a token/match memory** for incremental updates.

Conflict Resolution (Rule Selection Strategy)

When multiple rules are eligible to fire at the same time, conflict resolution strategies decide **which rule should be executed first**.

Conflict Set:
The set of all rules that are ready to fire at a given moment.

In rule-based systems, after matching (often using something like RETE), you might end up with multiple rules that are all eligible to fire at the same time—these form a "conflict set." Conflict resolution is the process of deciding which rule to execute first (or at all) when there's a tie. Without it, the system could behave unpredictably, like firing rules in random order leading to inconsistent results. It's part of the inference engine's job to prioritize based on strategies, ensuring logical and efficient reasoning.

Conflict Resolution Strategies

Conflict resolution is the process of deciding which rule to execute first (or at all) when there's a tie. Without it, the system could behave unpredictably, like firing rules in random order leading to inconsistent results. It's part of the inference engine's job to prioritize based on strategies, ensuring logical and efficient reasoning.

Key strategies:

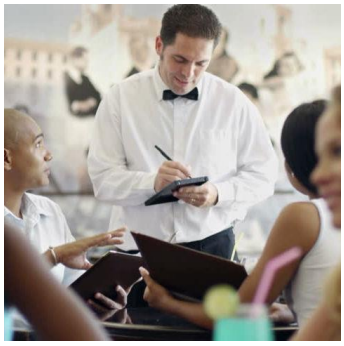
- **Recency**: Prioritizing newest information
- **Refractoriness**: Preventing Repetitive Actions
- **Specificity**: Prioritize the rule with the more specific or detailed conditions.
- **Priority** (or **Salience**): This is an explicit, pre-defined importance ranking given to rules by the system designer.
- **Depth** (or **FIFO** - First In, First Out): Prioritizes rules based on the order in which they were added
- **Breadth:** Advance all tasks a little bit.
- **Complexity/Simplicity:** Complexity First or Simplicity First based on the situation
- **Random:** "coin toss" of AI.
- **LEX (Lexicographic):** Using multiple criteria in a strict hierarchy, like words in a dictionary.
- **MEA** (Means-Ends Analysis): Goal-oriented optimization.

Let us consider a Restaurant Kitchen Example

First, imagine the kitchen's **"Rule Based AI System"** has a set of rules, like a chef's standard operating procedures. When new information arrives (a new order, an update), multiple rules might become applicable at the same time. This list of applicable rules is called the **Conflict Set**.
The "Conflict Resolution Strategy" is the head chef's method for deciding **which rule to execute first** from that conflict set.

- **Orders** represent rules that are ready to fire (matched conditions).
- **Kitchen rush hour** is the conflict set—multiple orders are eligible, but resources (like stoves, staff) are limited, so only one can be handled at a time.
- **Chef (inference engine)** decides the order of preparation to ensure efficiency, customer satisfaction, and logical flow (e.g., appetizers before mains).
- **Firing a rule** is like completing and serving an order.

This example highlights how strategies prioritize to avoid chaos, such as serving desserts before entrees or ignoring VIP requests.




Order tickets by Waiters


kitchen's physical order rail where tickets are clipped.

1. Recency: Prioritizing the Newest Information

Principle: Prioritize the rule that is triggered by the *most recently added or updated* piece of information.

Restaurant Example: The stream of order tickets coming from the waiter's station.

Step-by-Step Process:

Step 1: The Working Memory is Updated.
The "Working Memory" is like the kitchen's physical order rail where tickets are clipped. New facts are new tickets.

- **Ticket #101 (10:00 AM):** "Table 5: 2x Burgers (Medium), 1x Fries."
- **Ticket #102 (10:02 AM):** "Table 7: 1x Caesar Salad, 1x Soup."

Step 2: Rules are Matched (The Conflict Set).
The chef's "rules" are activated by these tickets.

- **Rule: Grill-Station** matches with Ticket #101 (because it has burgers).
- **Rule: Cold-Station** matches with Ticket #102 (because it has a salad).

Both rules are now in the "Conflict Set." Which one should the kitchen act on first?

Step 3: Applying the Recency Strategy.
The system looks at the timestamp of the facts that triggered each rule.

- Rule Grill-Station was triggered by a fact (Ticket #101) from **10:00 AM**.
- Rule Cold-Station was triggered by a fact (Ticket #102) from **10:02 AM**.

Step 4: The Decision.

Recency prioritizes the rule with the newest fact. Therefore, the Cold-Station rule (for the salad) fires first. The kitchen starts on Table 7's order because it's the most recent, ensuring the flow of work reflects the latest customer arrivals.

Why this is useful: It keeps the system responsive. If a new ticket comes in with an **"URGENT: Allergic Reaction, Modify Order!"** update, Recency ensures this critical new information is handled immediately, before continuing with older, less urgent tasks.

2. Refractoriness: Preventing Repetitive Actions

Principle: After a rule fires on a specific set of facts, it is temporarily disabled from firing again on the *exact same set of facts*. This prevents infinite loops and redundant work.

Restaurant Example: Marking a ticket as "COMPLETE" and removing it from the rail.

Step-by-Step Process:

Step 1: A Rule Fires Successfully.
Let's go back to **Ticket #101** for Table 5. The kitchen has finished cooking and serving the two burgers and fries.

Step 2: The Fact is Removed or Marked.
In a well-run system, the ticket is taken off the rail and thrown away (the fact is removed from Working Memory). Or, it's stamped "DONE" and moved to a completed orders pile (the fact is marked as processed and no longer eligible to trigger the *same* rule).

Step 3: A Potential Problem Arises (The "Glitch").
Now, imagine there's a software glitch, and the printer **re-prints Ticket #101** for Table 5. The exact same ticket—"2x Burgers (Medium), 1x Fries"—appears on the rail again. This is the AI equivalent of an unchanged fact sitting in working memory.

Step 4: Applying the Refractoriness Strategy.
Without Refractoriness, the Grill-Station rule would see this "new" ticket, match with it, and fire again. The kitchen would unnecessarily cook two more burgers and fries for a table that has already been served! This is an "infinite loop" of wasted effort.

Thanks to Refractoriness, the system "remembers" that the Grill-Station rule *just fired* for the *exact same set of facts* (the identical order details for Table 5). The rule is temporarily "refracted" or inactive for this specific data.

Step 5: The Decision.
The system **ignores the reprinted ticket**. It does not add the redundant order to the conflict set. The kitchen continues with its current work, avoiding a costly and confusing mistake.

3.Specificity

Principle: When two or more rules are in conflict, prioritize the rule with the **more specific or detailed conditions**. A rule is more specific if it has more conditions, or its conditions are a special case of a more general rule. This mimics an expert's thinking: "I have a general rule, but there's a special case that overrides it."

Restaurant Example: Handling a standard order vs. a highly customized or critical order (like an allergy).

Step-by-Step Process: The Conflicting Orders

Imagine two orders come in at the same time. The kitchen's "Working Memory" now contains these two facts:

- **Fact A (Ticket #201):** Order: Cheeseburger, Fries
- **Fact B (Ticket #202):** Order: Cheeseburger - NO ONIONS (Allergy), EXTRA PICKLES, Gluten-Free Bun, Side Salad instead of Fries

Now, let's define the kitchen's "Rules." These are the pre-programmed procedures for the staff.

Step 1: Rules are Matched (The Conflict Set is Created)
Both tickets trigger rules. The system identifies all applicable rules, creating the "Conflict Set."

- **Rule 1: General-Burger-Rule**
  - **IF:** Order contains a Burger
  - **THEN:** Assign to Grill Station with Standard Protocol
  - *This rule is very broad. It applies to ANY burger order.*
- **Rule 2: Specific-Allergy-Protocol**
  - **IF:** Order contains a Burger **AND** Order has an Allergy Alert **AND** Order has a Special Bun Request
  - **THEN:** Assign to Dedicated Allergy-Friendly Station, Sanitize Grill, Notify Head Chef
  - *This rule is much more detailed. It only applies to a very specific subset of burger orders.*
  -

Both Rule 1 and Rule 2 match the facts in the working memory. Specifically, both see the "Cheeseburger" in their orders. This creates a conflict. Which rule should fire first?

Step 2: Applying the Specificity Strategy
The system's "Head Chef" (the conflict resolver) now compares the rules. It doesn't look at timestamps; it looks at **how detailed the "IF" conditions are.**

- **Rule 1 (General)** has **1 condition**: Order contains a Burger.
- **Rule 2 (Specific)** has **3 conditions**: Burger + Allergy Alert + Special Bun Request.

The Key Insight: Rule 2 is a *special case* of Rule 1. Every order that matches Rule 2 (a burger with an allergy) will also match Rule 1 (a burger). However, the reverse is not true. A standard burger matches Rule 1 but not Rule 2.

Therefore, **Rule 2 is more specific** and is given priority.

Note: By prioritizing the specific rule, you enforce a clean workflow from the start:

1. The expeditor sees Ticket #202 and immediately initiates the Specific-Allergy-Protocol.
2. The prep cook is instructed to wash hands, put on clean gloves, and use only the dedicated ingredients *first*.
3. The safe meal is prepared and cooked without any prior contact with allergens.
4. *After* the critical order is secured and moving, the general rule for the standard burger is actioned. Any contamination from the standard ingredients stays within the "general" workflow.

Step 3: The Decision and Execution
The **Specificity-Strategy** resolves the conflict by selecting **Rule 2: Specific-Allergy-Protocol** to fire first.

What happens in the kitchen (The "THEN" actions):

1. Ticket #202 is prioritized.
2. The head chef is notified.
3. A specific, clean station is prepared to avoid cross-contamination.
4. The kitchen carefully follows the precise instructions (no onions, gluten-free bun).

Only after this specific, high-priority task is initiated would the system go back and fire **Rule 1** for the standard burger order (Ticket #201).

Why Specificity is Crucial: The "What If" Scenario
Let's see what would happen *without* the Specificity strategy. If the system used a simple "first-come, first-served" or arbitrary order:

1. **Rule 1 (General-Burger-Rule)** fires for Ticket #202.
2. The order is sent to the busy main grill.
3. The staff, following "standard protocol," might use a bun that has gluten or cook the burger where onions were just grilled.
4. **Result: A potentially life-threatening mistake.** The broad, general rule has caused harm because it ignored the critical nuances.

Specificity ensures that specialized knowledge (the exception) overrides the general default. It's the difference between a novice cook who just follows a basic recipe and an expert chef who knows that a customer's allergy is the most important piece of information on the ticket.

4.Priority / Salience

Principle: This is an **explicit, pre-defined importance ranking** given to rules by the system designer. Think of it as a "VIP pass" for certain rules. Unlike Recency (which is time-based) or Specificity (which is logic-based), Priority is based purely on **business policy or strategic importance**.

Restaurant Example: The head chef or manager has a policy that certain types of customers or orders *must* be prioritized, regardless of the order in which they came in or how simple they are.

Step-by-Step Process: The Saturday Night Rush

It's a busy Saturday night. The kitchen's "Working Memory" (the order rail) has three tickets, all received within a minute of each other. The timestamps are almost identical, so Recency isn't a strong differentiator.

- **Ticket #301 (10:05:00 PM):** Table 12: Pepperoni Pizza, Garlic Bread - *This is a regular table.*
- **Ticket #302 (10:05:15 PM):** TAKEOUT: Chicken Curry with Rice - *This is a takeout order.*
- **Ticket #303 (10:05:30 PM):** VIP TABLE (Mr. Rossi): Dry-Aged Steak, Truffle Mashed Potatoes - *Mr. Rossi is a famous food critic and a regular.*

Now, let's define the kitchen's "Rules." But this time, the head chef has assigned each rule a **SALIENCE** value (e.g., from 1-100, where 100 is highest priority).

Step 1: Rules are Matched with their Salience (The Annotated Conflict Set)
As tickets arrive, they activate rules. The system notes not just which rules are triggered, but also their pre-assigned salience.

- **Rule: Handle-Takeout-Order (Salience: 30)**
  - **IF:** Order Type is TAKEOUT
  - **THEN:** Assign to the secondary line. Package for delivery.
- **Rule: Handle-Regular-Order (Salience: 50)**
  - **IF:** Order Type is DINE-IN and Table Status is REGULAR
  - **THEN:** Assign to the main cooking line with standard priority.
- **Rule: Handle-VIP-Order (Salience: 90)**
  - **IF:** Order Type is DINE-IN and Table Status is VIP
  - **THEN:** Notify Head Chef personally. Use premium ingredients. Assign to the primary station with immediate priority.

All three rules are now in the "Conflict Set." The system's task is to decide the order of execution.

Step 2: Applying the Priority/Salience Strategy
The "Head Chef" (the conflict resolver) now performs a simple sort. It looks **only at the salience number** associated with each rule.
It creates a ranked list:

1. **Salience 90:** Handle-VIP-Order (for Ticket #303)
2. **Salience 50:** Handle-Regular-Order (for Ticket #301)
3. **Salience 30:** Handle-Takeout-Order (for Ticket #302)

The Key Insight: The decision is not based on logic, time, or detail. It's based purely on a **business rule**: "VIP customers always come first."

Step 3: The Decision and Execution
The **Priority-Strategy** resolves the conflict by selecting the rule with the highest salience to fire first.

What happens in the kitchen:

1. **FIRST:** Despite arriving last, **Ticket #303 (VIP Steak)** is processed immediately. The head chef personally oversees it. The best cook on the primary station starts the dish.
2. **SECOND:** Once the VIP order is underway, **Ticket #301 (Regular Pizza)** is handled by the main line.
3. **THIRD:** Finally, **Ticket #302 (Takeout Curry)** is prepared on the secondary line and boxed up.

Why Explicit Priority is Crucial: The "What If" Scenario

Let's see what would happen if the kitchen used a simple **First-In-First-Out (FIFO)** strategy instead of Salience.

1. The kitchen works on tickets in the order they arrived.
2. **Ticket #301 (Pizza)** is cooked and served. Table 12 is happy.
3. **Ticket #302 (Takeout)** is cooked and packaged. The delivery driver leaves.
4. **Finally, Ticket #303 (VIP Steak)** is started.

The Result: A Business Disaster.

- Mr. Rossi, the important critic, has been waiting for 25 minutes while simpler orders were completed.
- He has a terrible experience, writes a scathing review, and the restaurant's reputation suffers.
- The policy of "cherishing VIPs" has been violated by a blind adherence to arrival time.

Salience allows the system designer to **encode strategic knowledge directly into the system.** It overrides other strategies when business policy demands it.

5.Depth (via FIFO)

Principle: This strategy prioritizes rules based on the **order in which they were added to the "Conflict Set."** The rule that has been waiting the longest gets to fire first. It's a fair, queue-based system that ensures no task gets ignored or "starved." The term "Depth" comes from the idea of following one chain of reasoning to its conclusion before starting another, much like a depth-first search in a graph.

Restaurant Example: The classic order rail where tickets are clipped in the sequence they arrive. The chef works from left to right, always taking the oldest ticket first.

Step-by-Step Process: The Fair Queue

Imagine a steady stream of orders comes in. The kitchen's "Working Memory" is the order rail.

Step 1: Orders Arrive and are Queued (The FIFO Queue is Built)

The tickets are placed on the rail in the order they are received. This creates a literal "First-In, First-Out" (FIFO) queue.

- **Time 7:00 PM:** Ticket #41 - Table 3: 2x Spaghetti Bolognese -> **Added to the rail.**
- **Time 7:01 PM:** Ticket #42 - Table 5: 1x Caesar Salad -> **Added to the rail.**
- **Time 7:02 PM:** Ticket #43 - Table 7: 1x Grilled Chicken Sandwich -> **Added to the rail.**

The "Conflict Set" is the list of all tickets on the rail. The rule is simple: IF there is an order on the rail, THEN cook it.

Step 2: Applying the FIFO Strategy

The "Head Chef" (the conflict resolver) has one simple instruction: **Always pick the ticket on the far left (the one that has been there the longest).**

The system doesn't ask:

- Is this order for a VIP? (Ignores **Salience**)
- Did it just arrive? (Ignores **Recency**)
- Is it incredibly complex? (Ignores **Specificity**)

It only asks: **"Who's been waiting the longest?"**

Step 3: The Decision and Execution

The **FIFO Strategy** fires rules in strict chronological order.

1. **FIRST:** The chef takes **Ticket #41 (Table 3's Spaghetti)** and starts cooking. This order has "depth" because it was activated first.
2. **SECOND:** Once the spaghetti is served, the chef moves to the next oldest: **Ticket #42 (Table 5's Salad)**.
3. **THIRD:** Finally, the chef handles **Ticket #43 (Table 7's Sandwich)**.

Why this is useful (Preventing Starvation): Imagine if the system used a **LIFO (Last-In, First-Out)** strategy instead. Every time a new order came in, it would jump the queue. Ticket #41 for Table 3 would *never* be cooked because new tickets would constantly pre-empt it. The customers at Table 3 would wait forever—their order is "starved." FIFO guarantees fairness.

Step-by-Step Process: Depth in a Multi-Course Meal

Now, let's explore the "Depth" aspect, which is about following a sequence of related actions to completion. This is like a "rule chain."

Step 1: A Rule Fires, Creating New Facts

A single event can trigger a sequence of rules. Let's say the kitchen has a rule for managing multi-course meals.

- **Fact:** Ticket #50 - Table 9: 3-Course Meal (Appetizer, Main, Dessert)
- **Rule 1: Start-Meel-Sequence**
    - **IF:** Order is a Multi-Course Meal
    - **THEN:** 1. Fire the appetizer order. 2. Add a new fact to memory: "Table 9: Appetizer Served".

This rule fires (because it was the first on the rail), and its action adds a **new fact** to working memory.

Step 2: The New Fact Triggers the Next Rule in the Chain

The new fact "Table 9: Appetizer Served" now activates the next rule in the sequence.

- **Rule 2: Monitor-Appetier-Completion**
    - **IF:** "Table 9: Appetizer Served" is in memory **AND** 5 minutes have passed
    - **THEN:** 1. Fire the main course order. 2. Remove "Appetizer Served" fact. 3. Add "Table 9: Main Course Served".

Step 3: Applying the "Depth" Strategy

The "Depth" strategy prioritizes finishing this chain of reasoning for **Table 9** before fully engaging with a new, unrelated order from **Table 10**.

Here's the scenario:

- **Chain for Table 9 is active:** The appetizer has been served. The clock is ticking. In 5 minutes, the main course *must* be fired.
- **A new order arrives:** Ticket #51 - Table 10: 1x Soup (a very simple, quick dish).

The "Depth" Approach:

The system recognizes that it is "deep" in the process of serving Table 9. Even though the soup for Table 10 is quick, the main course for Table 9 is time-critical within its sequence. Therefore, the system will prioritize firing **Rule 2** to start Table 9's main course as soon as the 5 minutes are up, *before* starting Table 10's soup.

6. Breadth

Principle: This strategy prioritizes rules that are at the "beginning" of their reasoning chain or that represent independent, parallelizable tasks. Instead of following one complex task to completion (depth), it tries to advance **all tasks a little bit**. This is like breadth-first search in AI, where you explore all immediate options before going deeper into any one of them.

Restaurant Example: During a rush, the chef focuses on getting **something** out to every table quickly, rather than completing one table's entire meal while others wait with empty plates.

Step-by-Step Process: The Saturday Night Rush
It's 8:00 PM and the restaurant is full. Several orders hit the kitchen at once. The "Working Memory" (order rail) contains:

- **Ticket #71:** Table 10: 4-Course Meal (Salad, Soup, Steak, Dessert) - *A complex, deep order.*
- **Ticket #72:** Table 11: 2x Burgers with Fries - *A medium-complexity order.*
- **Ticket #73:** Table 12: 1x Caesar Salad - *A very simple, shallow order.*

Step 1: Rules are Matched (Identifying "Shallowness")
Each ticket triggers the rule IF there is an order, THEN start it. But the **Breadth** strategy analyzes more than just the rule—it analyzes the **entire task chain** that the rule initiates.

- **Task Chain for Ticket #71 (Table 10):** Make Salad -> Make Soup -> Cook Steak -> Prepare Dessert. This is a **deep, sequential chain**.
- **Task Chain for Ticket #72 (Table 11):** Cook Burgers -> Fry Fries. This is a **medium-depth chain** with some parallelism.
- **Task Chain for Ticket #73 (Table 12):** Toss Salad. This is a **very shallow, single-step task**.

Step 2: Applying the Breadth Strategy
The "Head Chef" (the conflict resolver) asks: **"Which task can I complete quickly to give the most tables some progress right now?"**
The strategy identifies the "shallowest" activations—the tasks that are nearest to completion or require the fewest steps.

- **Ticket #73 (Caesar Salad)** is the shallowest. It's one quick step.
- **Ticket #71's Salad** is also a shallow step within a deep chain.
- **Ticket #72's Burger patty preparation** is another starting point.

The Breadth Approach: Instead of diving deep into Table 10's four-course meal, the chef will prioritize starting (or completing) the quick, independent tasks **across all orders.**

Step 3: The Decision and Execution
The **Breadth Strategy** creates an execution plan that advances all tables simultaneously.

1. **FIRST ACTION:** The chef immediately prepares and sends out **Ticket #73: Table 12's Caesar Salad**. This table is now happy and served.
2. **SECOND ACTION:** The chef starts the **shallowest step of the deep order:** They quickly prepare the **green salad for Ticket #71 (Table 10)** and send it out. Table 10 now has their first course and feels the meal has started.
3. **THIRD ACTION:** The chef now starts the **burger patties for Ticket #72 (Table 11)**, a medium-depth task.

The Outcome: Within the first 10 minutes:

- Table 12 is completely finished and satisfied.
- Table 10 has their first course and is engaged.
- Table 11's order is underway.

This is far better than the **Depth** approach, which would have meant Table 10 gets all four courses while Tables 11 and 12 wait with nothing.

Why Breadth is Crucial: The "What If" Scenario (Breadth vs. Depth)
Let's see what happens if the kitchen used a **Depth-First** strategy on the same set of orders.

1. **Depth-First Decision:** Start the deepest chain first—Table 10's four-course meal.
2. **Execution:** The kitchen focuses entirely on Table 10.
3. **Time 8:15:** Table 10 receives their salad. Tables 11 and 12 have nothing.
4. **Time 8:25:** Table 10 receives their soup. Tables 11 and 12 are still waiting, getting impatient.
5. **Time 8:45:** Table 10 receives their steak. The customers at Table 12 (who only wanted a simple salad) have been waiting for 45 minutes and are furious. Table 11 is also complaining.

The Result: While one table is perfectly served, two other tables have a terrible experience. The **overall customer satisfaction** in the restaurant is low.

Breadth solves this by maximizing overall progress. It's the philosophy that it's better for everyone to be a little happy than for one person to be completely happy while others are completely ignored.

7. Complexity/Simplicity

Principle: This strategy prioritizes rules based on how complex or simple they are to execute.

- **Complexity-First:** Tackles the most demanding rules first, assuming they're critical path items that could bottleneck the system.
- **Simplicity-First:** Handles the easiest rules first to quickly clear the queue and build momentum.

Restaurant Example: The chef must decide whether to tackle the complicated special order that requires special attention OR knock out several simple side dishes first to clear the order rail.

Step-by-Step Process

Scenario-1: Complexity-First Strategy

When Quality and Critical Path Dominate

Imagine it's a normal dinner service. The kitchen has time to focus on doing complex dishes perfectly. The order rail shows:

- **Ticket #81:** Table 4: 1x Side of Fries - *Very simple (2 conditions: cook fries, season)*
- **Ticket #82:** Table 6: 1x "Chef's Special" Multi-Step Dish - *Extremely complex (10+ conditions: marinate, sear, roast, make sauce, garnish, etc.)*

Step 1: Analyzing Rule Complexity
The system evaluates the "cognitive load" or "execution steps" each rule requires.

- **Rule: Make-Fries (Low Complexity)**
  - **IF:** Order contains Fries
  - **THEN:** Drop fries in fryer, wait 3 minutes, season, serve. → **Few, quick, non-blocking steps.**
- **Rule: Make-Chefs-Special (High Complexity)**
  - **IF:** Order contains "Chef's Special" **AND** Protein is at temperature **AND** Sauce ingredients are prepped **AND** Garnish is ready...
  - **THEN:** Sear protein, make reduction sauce, coordinate side dishes, plate artistically... → **Many, time-consuming, interdependent steps.**

Step 2: Applying the Complexity-First Strategy
The "Head Chef" asks: **"Which task, if delayed, will cause the biggest backup or require the most uninterrupted focus?"**
The answer is the complex dish. It requires the head chef's full attention and multiple stations. If they start the fries first, the complex dish gets pushed back, potentially ruining the timing for Table 6's entire meal.

Step 3: The Decision and Execution
The **Complexity-First Strategy** prioritizes the rule with the most conditions and steps.

1. **FIRST:** The chef starts **Ticket #82 (Complex Special)**. They begin the multi-step process: searing the meat, starting the sauce reduction, and coordinating with other stations.
2. **SECOND:** While the special is roasting or reducing (steps with waiting time), the chef uses the **downtime** to quickly handle **Ticket #81 (Fries)**. The fries cook in 3 minutes and are served.
3. **Result:** The complex dish is started early, ensuring it gets the focus it needs. The simple task is completed during the natural pauses in the complex task.

Why Complexity-First Works Here: It prevents the "last-minute rush" on complex items. If you save the complex dish for last, you might end up with several complex orders backing up simultaneously, overwhelming the kitchen.

Scenario-2: Complexity-First Strategy

When Throughput and Momentum Are Key
Now imagine it's the peak of the Saturday night rush. The kitchen is overwhelmed. The order rail is packed:

- **Ticket #91:** Table 2: 1x Side Salad - *Extremely simple (30 seconds)*
- **Ticket #92:** Table 3: 1x Soft Drink - *Extremely simple (10 seconds)*
- **Ticket #93:** Table 5: 4x Customized "Well-Done" Steaks with Different Sides - *Very complex (15+ minutes)*

Step 1: Analyzing Rule Simplicity
The system identifies which rules can be completed almost instantly.

- **Rule: Make-Side-Salad / Pour-Drink (High Simplicity)**
    - **IF:** Order contains Salad OR Order contains Drink
    - **THEN:** Toss greens / Pour drink → Serve. → **Near-instantaneous completion.**
- **Rule: Cook-Complex-Steak-Order (High Complexity)**
    - **IF:** Order contains multiple customized steaks...
    - **THEN:** Long, resource-intensive process. → **Major time investment.**

Step 2: Applying the Simplicity-First Strategy
The "Head Chef" asks: **"How can I make the most visible progress right now and free up mental space?"**
The answer is to clear the easy tasks. The complex steaks will take time no matter what. By quickly handling the salad and drink, the chef achieves two things:

1. **Reduces the queue length visibly** (from 3 tickets to 1).
2. **Frees up cognitive load** – the simple tasks are no longer taking up "mental RAM."

Step 3: The Decision and Execution
The **Simplicity-First Strategy** prioritizes the rules with the fewest conditions and quickest execution.

1. **FIRST:** A staff member pours **Ticket #92 (Drink)** and runs it to Table 3. ✅ One ticket done.
2. **SECOND:** The chef quickly assembles **Ticket #91 (Side Salad)** and sends it out. ✅ Second ticket done.
3. **THIRD:** Now, with a clearer rail and fewer distractions, the chef focuses entirely on **Ticket #93 (Complex Steaks)**.

Why Simplicity-First Works Here: It creates a psychological and practical advantage. The kitchen feels less overwhelmed, and the staff builds momentum. Customers see items arriving quickly, which improves their perception of service, even if their main course is still cooking.

8. Random Selection

Principle: When multiple rules are in conflict, this strategy selects one **at random** using a random number generator or similar chance-based mechanism. It introduces non-determinism into the system, meaning the same situation might be resolved differently each time.

Restaurant Example: The chef, when faced with two orders that are truly identical in every way, closes their eyes and picks one at random.

Step-by-Step Process: The Identical Orders Dilemma
It's a busy but manageable evening. The kitchen's order rail has two tickets that arrived at virtually the same time.

- **Ticket #101:** Table 5: 1x Classic Cheeseburger, No modifications - *Received at 7:30:00 PM*
- **Ticket #102:** Table 6: 1x Classic Cheeseburger, No modifications - *Received at 7:30:02 PM*

Step 1: Rules are Matched - A Perfect Conflict
Both tickets trigger the exact same rule:

- **Rule: Cook-Standard-Burger**
    - **IF:** Order is "Classic Cheeseburger"
    - **THEN:** Grill patty, toast bun, assemble.

The "Conflict Set" now contains two activations of the *same rule* for two *different but identical* orders.

Step 2: Applying Other Strategies (Why Random is Needed)

The chef tries to use other strategies, but they fail to break the tie:

- **Recency:** The timestamps are only 2 seconds apart. Is Table 6's order *meaningfully* more recent? Not really. It's a tie for all practical purposes.
- **Specificity:** Both orders have the exact same level of detail. They are equally specific.
- **Salience/Priority:** Both are for regular tables. No VIP priority.
- **Depth/FIFO:** Table 5's ticket was first. But should Table 6 always be penalized for being two seconds later?
- **Breadth/Complexity:** Both orders are equally simple and belong to the same "level" of task.

The kitchen faces a true deadlock. There is no logical, business-policy, or timing-based reason to choose one over the other.

Step 3: Applying the Random Strategy
Instead of overthinking, the chef employs a random selection method. They might:

- **Flip a coin:** Heads for Table 5, Tails for Table 6.
- **Pick the nearest ticket:** Without looking, grab the ticket closest to hand.
- **Use a random number generator:** The AI equivalent.

Let's say the chef flips a coin. It comes up **Tails**.

Step 4: The Decision and Execution
The **Random Strategy** resolves the conflict by chance.

1. **FIRST:** The chef starts cooking the burger for **Table 6 (Ticket #102)** because the coin toss selected it.
2. **SECOND:** Immediately after, the chef starts the burger for **Table 5 (Ticket #101)**.

The total difference in waiting time between the two tables will be negligible—just the few minutes it takes to cook one burger.

Why a Random Strategy is Useful: The "What If" Scenario
What if the kitchen *always* used **FIFO (First-In, First-Out)** in this scenario?

- Table 5's order would **always** be cooked before Table 6's.
- If Table 6 is a regular customer, they might subconsciously notice they *always* get their food after Table 5, leading to a perception of unfairness, even if the time difference is small.

The Random strategy prevents systematic bias. Over the course of a week, Table 5 and Table 6 will each have their order first about 50% of the time. This is the essence of fairness when no other distinguishing factors exist.

9. LEX (Lexicographic) Strategy: The Multi-Level Tie-Breaker

Principle: LEX sorts rules using multiple criteria in a strict hierarchy, like words in a dictionary (hence "lexicographic"). It applies the first criterion to all rules, then the second to break ties, then the third, and so on.

Restaurant Example: The chef has a strict priority list for sorting tickets: **1) VIP status, 2) Arrival time, 3) Table number.**

Step-by-Step Process: The Saturday Night Dilemma

Three orders arrive in quick succession. The working memory contains:

- **Ticket #301:** Table 8 (Regular), 7:05 PM, Burger
- **Ticket #302:** Table 5 (VIP), 7:07 PM, Steak

- **Ticket #303:** Table 3 (VIP), 7:06 PM, Salmon

Step 1: Define the Criteria Hierarchy
The restaurant's LEX policy is:
1. **Criterion 1: Customer Status** (VIP > Regular)
2. **Criterion 2: Arrival Time** (Earliest First)
3. **Criterion 3: Table Number** (Lowest First)

Step 2: Apply Criterion 1 (VIP Status)
The system groups tickets by the most important criterion:
- **VIP Group:** Ticket #302 (Table 5), Ticket #303 (Table 3)
- **Regular Group:** Ticket #301 (Table 8)

Result after Criterion 1: All VIP tickets are prioritized over regular tickets. Ticket #301 (Regular) is moved to the bottom of the list. The conflict is now between the two VIP tickets.

Step 3: Apply Criterion 2 (Arrival Time) to Break the Tie
The system looks at the arrival times of the VIP tickets:
- Ticket #303 (Table 3): 7:06 PM
- Ticket #302 (Table 5): 7:07 PM

Result after Criterion 2: The earliest VIP ticket (Table 3) is prioritized. The tie is broken.

Step 4: Apply Criterion 3 (If needed)
Criterion 3 (Table Number) wasn't needed here because Criterion 2 already broke the tie. If both VIP tickets had arrived at exactly the same time, the system would then pick the one with the lower table number.
Final LEX Order:
1. **Ticket #303 (Table 3 - VIP, 7:06 PM)**
2. **Ticket #302 (Table 5 - VIP, 7:07 PM)**
3. **Ticket #301 (Table 8 - Regular, 7:05 PM)**

Notice that the regular ticket (7:05 PM) arrived earlier than one VIP ticket (7:07 PM), but the VIP status criterion overrode recency. This is the essence of LEX—**a predictable, multi-level sorting system.**

10. MEA (Means-Ends Analysis): The Goal-Oriented Strategist

Principle: MEA is a goal-driven strategy. It evaluates each possible action based on how effectively it reduces the "difference" between the current state and a desired goal state. It chooses the rule that gets the system closest to its objective.

Restaurant Example: The head chef's goal is to "Have all customers served within 30 minutes of ordering." MEA will prioritize the order that best helps achieve this goal right now.

Step-by-Step Process: The 8:00 PM Rush
The kitchen is in a state of chaos. The goal is to **"Complete all orders within 30 minutes."** The current state is:
- **Ticket #401 (Table 10):** Burger & Fries - *Ordered 25 minutes ago. Fries are done, burger is 2 minutes from being ready.*
- **Ticket #402 (Table 11):** Well-Done Steak - *Ordered 10 minutes ago. Not started.*
- **Ticket #403 (Table 12):** Salad - *Ordered 5 minutes ago. Not started.*

Step 1: Define the Goal State
The **Ends** (Goal): All orders completed within 30-minute deadline.
The **Means** (Available Actions): Fire rules to cook the burger, steak, or salad.

Step 2: Analyze the "Distance to Goal" for Each Option
MEA calculates which action most effectively reduces the distance to the goal.

- **Option A: Finish Ticket #401 (Burger)**
  - **Current State:** This order is about to breach the 30-minute deadline (only 5 minutes left).
  - **Action:** Apply the rule to complete the burger.
  - **Distance Reduced: HIGH.** This action directly saves an order from being late. It's a critical intervention.
  - **New State:** One order completed on time, crisis averted.
- **Option B: Start Ticket #402 (Steak)**
  - **Current State:** The steak has 20 minutes left on its deadline, but it takes 15 minutes to cook.
  - **Action:** Start the well-done steak now.
  - **Distance Reduced: MEDIUM.** This is proactive and necessary, but not as urgent as saving an imminent failure.
  - **New State:** The steak is on track, but the burger is now late.
- **Option C: Start Ticket #403 (Salad)**
  - **Current State:** The salad has 25 minutes left and takes 3 minutes to make.
  - **Action:** Make the quick salad.
  - **Distance Reduced: LOW.** While easy, this does nothing to address the pressing deadline of the burger. It's a distraction from the main goal.
  - **New State:** A simple task is done, but the burger is late, and the steak is now tighter on time.

Step 3: Apply the MEA Strategy - Choose the Best "Means" to the "Ends"
The MEA strategy selects the rule that **maximizes the reduction in "distance to goal."**
Decision: The chef chooses **Option A: Finish Ticket #401**.
Why? Because it has the highest leverage on the primary goal of meeting deadlines. Letting an order go late is a bigger failure than slightly delaying the start of another.

Step 4: Execute and Re-evaluate
After the burger is served (on time!), the system re-evaluates:

- **New Goal:** "Complete the remaining orders within their deadlines."
- **New State:** Steak has 18 minutes left, Salad has 22 minutes left.
- **New MEA Analysis:** Now, starting the steak (which takes 15 minutes) becomes the most critical action to avoid a future deadline breach. The salad can still be made quickly later.

Why MEA is Powerful: The "What If" Scenario

Without MEA, a kitchen might use a **Simplicity-First** strategy and make the salad first because it's easy. This feels productive but is actually counter-productive. The burger becomes late, and the steak's timeline becomes dangerously tight.
MEA ensures that every decision is evaluated against the **overarching objective**, not just local efficiency.