UNIT-3 LOGICAL AGENT

Logical agents are a type of intelligent agent in artificial intelligence (AI) that use formal logic to represent knowledge and make decisions. These agents rely on logical reasoning to infer new knowledge from existing information and to determine the best actions to achieve their goals.

Definition

A logical agent is an AI system that employs formal logic as the basis for representing knowledge, reasoning, and decision-making. Logical agents utilize symbolic representations and logical operations to derive conclusions and make decisions.

In AI, knowledge-based agents use a process of reasoning over an internal representation of knowledge to decide what actions to take.

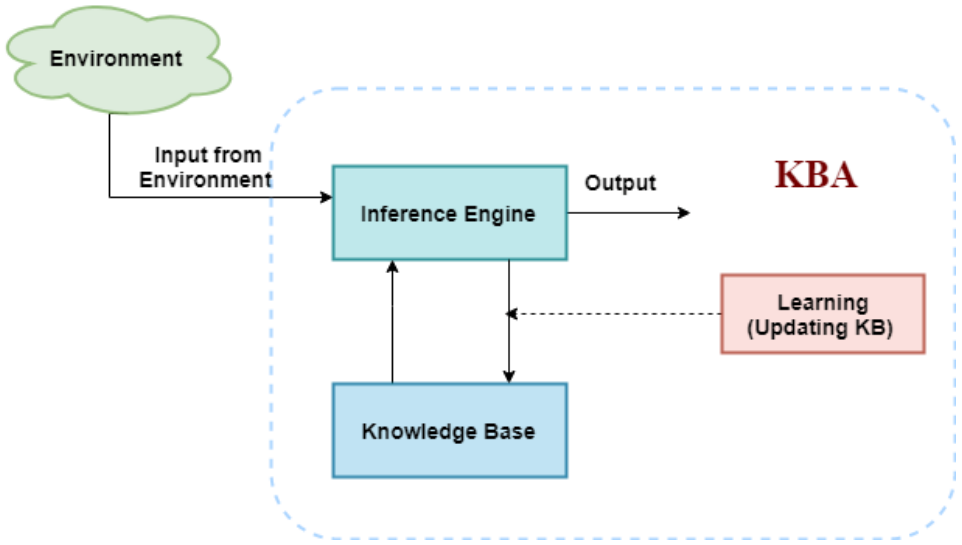
Knowledge-Based Agent

- An intelligent agent needs knowledge about the real world for taking decisions and reasoning to act efficiently.
- Knowledge-based agents are those agents who have the capability of maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently.
- Knowledge-based agents are composed of two main parts:
  - Knowledge-base and
  - Inference system.

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

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

The architecture of knowledge-based agent:



The above diagram is representing a generalized architecture for a knowledge-based agent. The knowledge-based agent (KBA) take input from the environment by perceiving the environment. The input is taken by the inference engine of the agent and which also communicate with KB to decide as per the knowledge store in KB. The learning element of KBA regularly updates the KB by learning new knowledge.

Knowledge base: Knowledge-base is a central component of a knowledge-based agent, it is also known as KB. It is a collection of sentences (here 'sentence' is a technical term and it is not identical to sentence in English). These sentences are expressed in a language which is called a knowledge representation language. The Knowledge-base of KBA stores fact about the world.

Why use a knowledge base?

Knowledge-base is required for updating knowledge for an agent to learn with experiences and take action as per the knowledge.

Inference system

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

Inference system generates new facts so that an agent can update the KB. An inference system works mainly in two rules which are given as:

- **Forward chaining**
- **Backward chaining**

Operations Performed by KBA

Following are three operations which are performed by KBA in order to show the intelligent behavior:

1. TELL: This operation tells the knowledge base what it perceives from the environment.
2. ASK : This operation asks the knowledge base what action it should perform.
3. Perform: It performs the selected action.A generic knowledge-based agent:

Following is the structure outline of a generic knowledge-based agents program:

```
1. function KB-AGENT(percept):
2. persistent: KB, a knowledge base
3.         t, a counter, initially 0, indicating time
4. TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
5. Action = ASK(KB, MAKE-ACTION-QUERY(t))
6. TELL(KB, MAKE-ACTION-SENTENCE(action, t))
7.  t = t + 1
8.  return action
```

The knowledge-based agent takes percept as input and returns an action as output. The agent maintains the knowledge base, KB, and it initially has some background knowledge of the real world. It also has a counter to indicate the time for the whole process, and this counter is initialized with zero.

Each time when the function is called, it performs its three operations:

- Firstly it TELLs the KB what it perceives.
- Secondly, it asks KB what action it should take
- Third agent program TELLS the KB that which action was chosen.

The MAKE-PERCEPT-SENTENCE generates a sentence as setting that the agent perceived the given percept at the given time.

The MAKE-ACTION-QUERY generates a sentence to ask which action should be done at the current time.

MAKE-ACTION-SENTENCE generates a sentence which asserts that the chosen action was executed.

Various levels of knowledge-based agent:

A knowledge-based agent can be viewed at different levels which are given below:

1. Knowledge level

Knowledge level is the first level of knowledge-based agent, and in this level, we need to specify what the agent knows, and what the agent goals are. With these specifications, we can fix its behavior. For example, suppose an automated taxi agent needs to go from a station A to station B, and he knows the way from A to B, so this comes at the knowledge level.

2. Logical level:

At this level, we understand that how the knowledge representation of knowledge is stored. At this level, sentences are encoded into different logics. At the logical level, an encoding of knowledge into logical sentences occurs. At the logical level we can expect to the automated taxi agent to reach to the destination B.

3. Implementation level:

This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

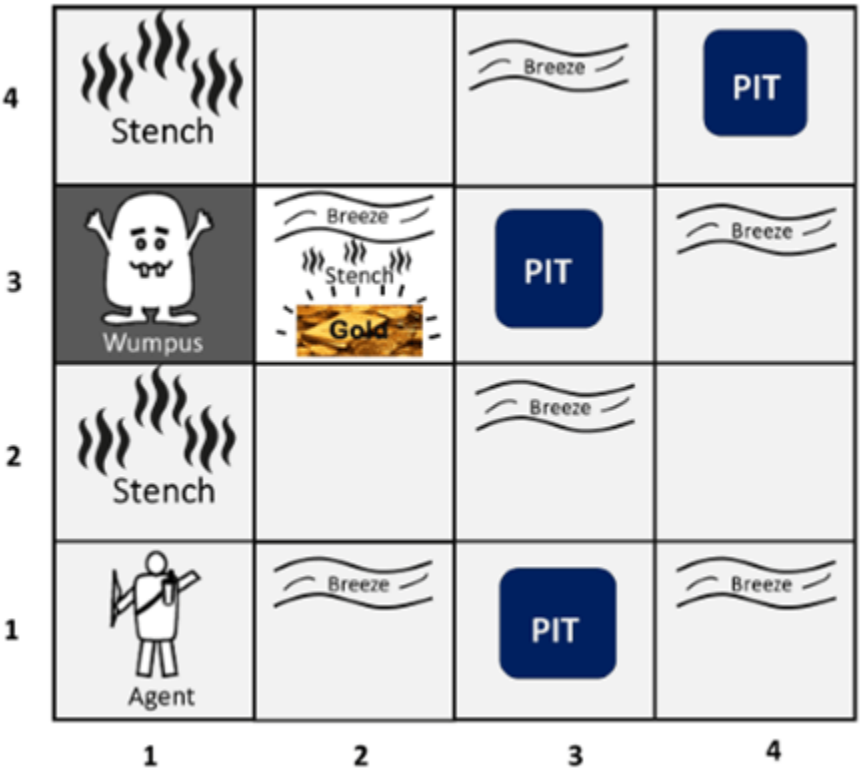
The Wumpus World

Wumpus world:

The Wumpus world is a simple world example to illustrate the worth of a knowledge-based agent and to represent knowledge representation. It was inspired by a video game **Hunt the Wumpus** by Gregory Yob in 1973.

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

Following is a sample diagram for representing the Wumpus world. It is showing some rooms with Pits, one room with Wumpus and one agent at (1, 1) square location of the world.



A typical wumpus world. The agent is in the bottom left corner, facing east (rightward).

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

a. The rooms adjacent to the Wumpus room are smelly, so that it would have some stench.
b. The room adjacent to PITs has a breeze, so if the agent reaches near to PIT, then he will perceive the breeze.
c. There will be glitter in the room if and only if the room has gold.
d. The Wumpus can be killed by the agent if the agent is facing to it, and Wumpus will emit a horrible scream which can be heard anywhere in the cave.

PEAS description of Wumpus world:

To explain the Wumpus world we have given PEAS description as below:

Performance measure:

- +1000 reward points if the agent comes out of the cave with the gold.
- -1000 points penalty for being eaten by the Wumpus or falling into the pit.
- -1 for each action, and -10 for using an arrow.
- The game ends if either agent dies or came out of the cave.

Environment:

- A 4*4 grid of rooms.
- The agent initially in room square [1, 1], facing toward the right.
- Location of Wumpus and gold are chosen randomly except the first square [1,1].
- Each square of the cave can be a pit with probability 0.2 except the first square.

Actuators:

- Left turn,
- Right turn
- Move forward
- Grab

- Release
- Shoot.

Sensors:

- The agent will perceive the **stench** if he is in the room adjacent to the Wumpus. (Not diagonally).
- The agent will perceive **breeze** if he is in the room directly adjacent to the Pit.
- The agent will perceive the **glitter** in the room where the gold is present.
- The agent will perceive the **bump** if he walks into a wall.
- When the Wumpus is shot, it emits a horrible **scream** which can be perceived anywhere in the cave.
- These percepts can be represented as five element list, in which we will have different indicators for each sensor.
- Example if agent perceives stench, breeze, but no glitter, no bump, and no scream then it can be represented as: **[Stench, Breeze, None, None, None]**.

The Wumpus world Properties:

o Partially observable: The Wumpus world is partially observable because the agent can only perceive the close environment such as an adjacent room.

o Deterministic: It is deterministic, as the result and outcome of the world are already known.

o Sequential: The order is important, so it is sequential.

o Static: It is static as Wumpus and Pits are not moving.

o Discrete: The environment is discrete.

- o One agent: The environment is a single agent as we have one agent only and Wumpus is not considered as an agent. Exploring the Wumpus world:

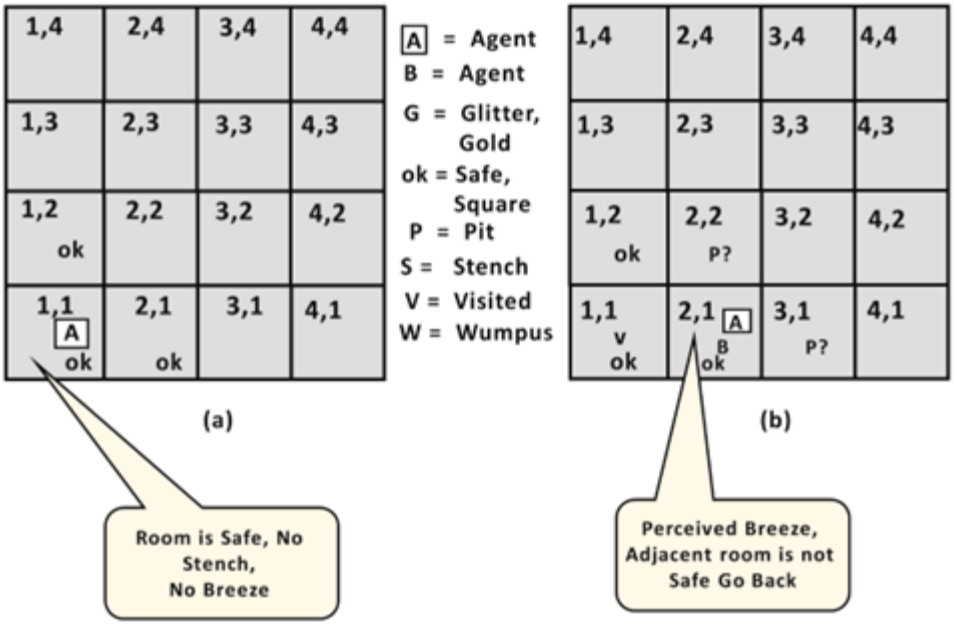
Now we will explore the Wumpus world and will determine how the agent will find its goal by applying logical reasoning.

Agent's First step:

Initially, the agent is in the first room or on the square [1,1], and we already know that this room is safe for the agent, so to represent on the below diagram

(a) that room is safe we will add symbol OK. Symbol A is used to represent agent, symbol B for the breeze, G for Glitter or gold, V for the visited room, P for pits, W for Wumpus.

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



The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [None,None,None,None,None]. (b) After moving to [2,1] and perceiving [None,Breeze,None,None,None].

Agent's second Step:

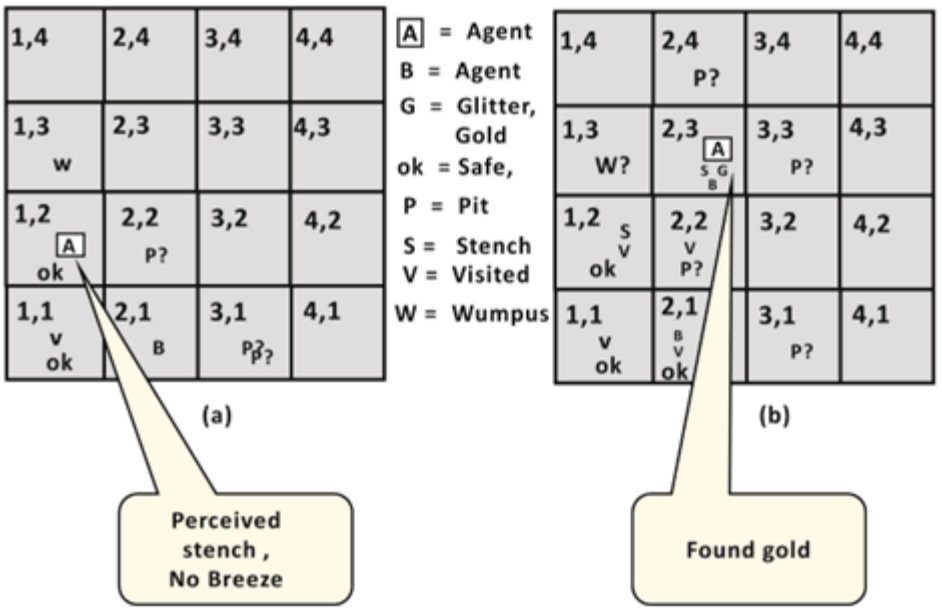
Now agent needs to move forward, so it will either move to [1, 2], or [2,1]. Let's suppose agent moves to the room [2, 1], at this room agent perceives some breeze which means Pit is around this room. The pit can be in [3, 1], or [2,2], so we will add symbol P? to say that, is this Pit room?

Now agent will stop and think and will not make any harmful move. The agent will go back to the [1, 1] room. The room [1,1], and [2,1] are visited by the agent, so we will use symbol V to represent the visited squares.

Agent's third step:

At the third step, now agent will move to the room [1,2] which is OK. In the room [1,2] agent perceives a stench which means there must be a Wumpus nearby. But Wumpus cannot be in the room [1,1] as by rules of the game, and also not in [2,2] (Agent had not detected any stench when he was at [2,1]). Therefore agent infers that Wumpus is in the room [1,3], and in current state,

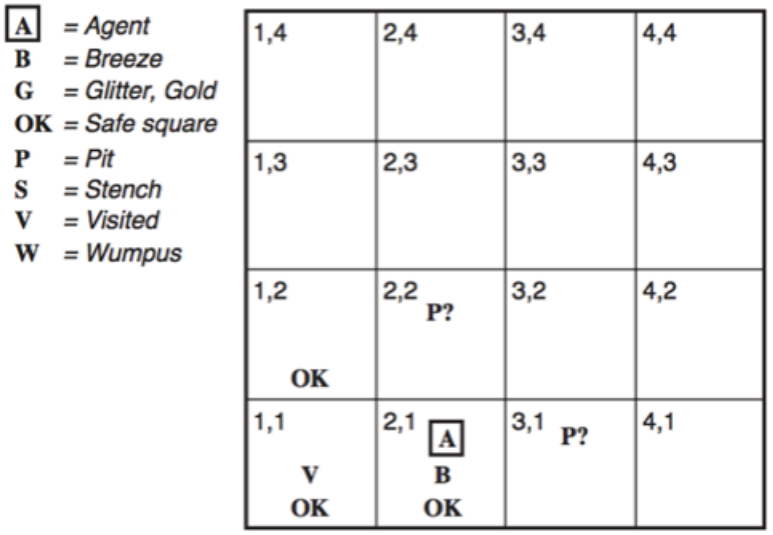
there is no breeze which means in [2,2] there is no Pit and no Wumpus. So it is safe, and we will mark it OK, and the agent moves further in [2,2].



Two later stages in the progress of the agent. (a) After moving to [1,1] and then [1,2], and perceiving [Stench,None,None,None,None]. (b) After moving to [2,2] and then [2,3], and perceiving [Stench,Breeze,Glitter,None,None].

Agent's fourth step:

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



LOGIC

A representation language is defined by its syntax, which specifies the structure of sentences, and its semantics, which defines the truth of each sentence in each possible world or model.

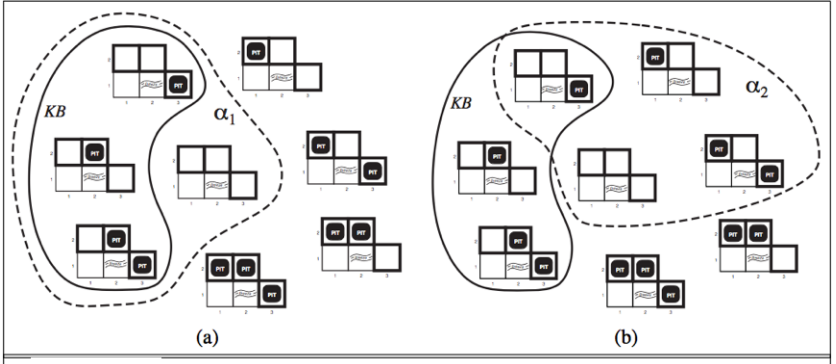
Syntax: The sentences in KB are expressed according to the syntax of the representation language, which specifies all the sentences that are well formed.

Semantics: The semantics defines the truth of each sentence with respect to each possible world.

Models: We use the term model in place of "possible world" when we need to be precise. Possible world might be thought of as (potentially) real environments that the agent might or might not be in, models are mathematical abstractions, each of which simply fixes the truth or falsehood of every relevant sentences.

If a sentence α is true in model m, we say that m satisfies α, or m is a model of α. Notation M(α) means the set of all models of α.

The relationship of entailment between sentence is crucial to our understanding of reasoning. A sentence α entails another sentence β if β is true in all world where α is true. Equivalent definitions include the validity of the sentence $\alpha \Rightarrow \beta$ and the unsatisfiability of sentence $\alpha \wedge \neg \beta$.
Logical entailment: The relation between a sentence and another sentence that follows from it.
Mathematical notation: $\alpha \models \beta$: αentails the sentence β.
Formal definition of entailment:
$\alpha \models \beta$ if and only if $M(\alpha) \subseteq M(\beta)$
i.e. $\alpha \models \beta$ if and only if, in every model in which αis true, β is also true.
(Notice: if $\alpha \models \beta$, then α is a stronger assertion than β: it rules out more possible worlds.)
Logical inference: The definition of entailment can be applied to derive conclusions.



Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

The KB is false in models that contradict what the agent knows. (e.g. The KB is false in any model in which [1,2] contains a pit because there is no breeze in [1, 1]).
Consider 2 possible conclusions α1and α2
We see: in every model in which KB is true,*a1* is also true. Hence KB⊨*a1*, so the agent can conclude that there is no pit in [1, 2].
We see: in some models in which KB is true,*a2* is false. Hence KB⊭*a2*, so the agent cannot conclude that there is no pit in [1, 2].

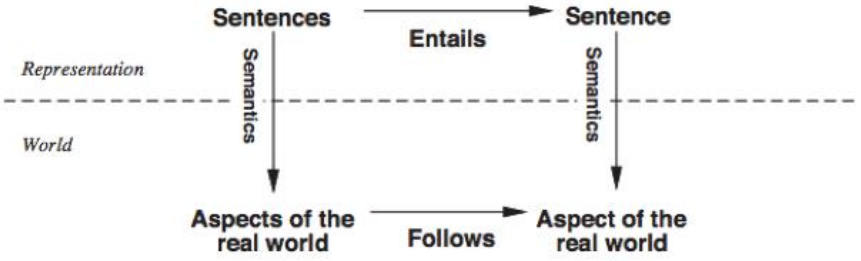
The inference algorithm used is called model checking: Enumerate all possible models to check that α is true in all models in which KB is true,
i.e. M(KB) ⊆ M(α).

If an inference algorithm i can derive α from KB, we write $KB \vDash_i \alpha$, pronounced as “α is derived from KB by i” or “i derives α from KB.”

Sound/truth preserving: An inference algorithm that derives only entailed sentences. Soundness is a highly desirable property.
(e.g. model checking is a sound procedure when it is applicable.)
Completeness: An inference algorithm is complete if it can derive any sentence that is entailed. Completeness is also a desirable property.

Inference is the process of deriving new sentences from old ones. Sound inference algorithms derive only sentences that are entailed; complete algorithms derive all sentences that are entailed.

If KB is true in the real world, then any sentence α derived from KB by a sound inference procedure is also true in the real world.



Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

PROPOSITIONAL LOGIC

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

A Very Simple Logic

Example:

a) It is Sunday.
b) The Sun rises from West (False proposition)
c) 3+3= 7(False proposition)
d) 5 is a prime number.

Following are some basic facts about propositional logic:

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and logical connectives.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called tautology, and it is also called a valid sentence.
- A proposition formula which is always false is called Contradiction.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "Where is Rohini", "How are you", "What is your name", are not propositions.

Syntax of propositional logic:

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

1. Atomic Propositions
2. Compound propositions

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

Example:

a) 2+2 is 4, it is an atomic proposition as it is a **true** fact.
b) "The Sun is cold" is also a proposition as it is a **false** fact.

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

Example:

1. a) "It is raining today, and street is wet."
2. b) "Ankit is a doctor, and his clinic is in Mumbai."

Logical Connectives:

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

1. Negation: A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
2. Conjunction: A sentence which has ∧ connective such as, P ∧ Q is called a conjunction.
   Example: Rohan is intelligent and hardworking. It can be written as,
   P= Rohan is intelligent,
   Q= Rohan is hardworking. → P∧ Q.
3. Disjunction: A sentence which has ∨ connective, such as P ∨ Q. is called disjunction, where P and Q are the propositions.
   Example: "Ritika is a doctor or Engineer",

Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as P ∨ Q.

4. Implication: A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as
   If it is raining, then the street is wet.
   Let P= It is raining, and Q= Street is wet, so it is represented as P → Q
5. Biconditional: A sentence such as P⇔ Q is a Biconditional sentence, example If I am breathing, then I am alive
   P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

Following is the summarized table for Propositional Logic Connectives:

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

Truth Table:

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is

called Truth table. Following are the truth table for all logical connectives:

For Negation:

P	¬ P
True	**False**
False	**True**

For Conjunction:

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

For disjunction:

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

For Implication:

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

Logical equivalence:

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

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

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

Limitations of Propositional logic:

We cannot represent relations like ALL, some, or none with propositional logic. Example:

a. All the girls are intelligent.

b. Some apples are sweet.

- Propositional logic has limited expressive power.
- In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

FIRST-ORDER LOGIC IN ARTIFICIAL INTELLIGENCE

In the topic of Propositional logic, we have seen that how to represent statements using propositional logic. But unfortunately, in propositional logic, we can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power. Consider the following sentence, which we cannot represent using PL logic. "Some humans are intelligent", or "Sachin likes cricket."

First-Order logic (FOL):

- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
- FOL is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as **Predicate logic or First-order predicate logic**. First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.
- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:
    - **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, ...
    - **Relations: It can be unary relation such as:** red, round, is adjacent, **or n-any relation such as:** the sister of, brother of, has color, comes between
    - **Function:** Father of, best friend, third inning of, end of, ...
- As a natural language, first-order logic also has two main parts:
    a. **Syntax**
    b. **Semantics**

Syntax of First-Order logic:

The syntax of FOL determines which collection of symbols is a logical expression in first-order logic. The basic syntactic elements of first-order logic are symbols. We write statements in short-hand notation in FOL.

Basic Elements of First-order logic:

Following are the basic elements of FOL syntax:

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

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2, ..., term n)**.

Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay). Chinky is a cat: => cat (Chinky).

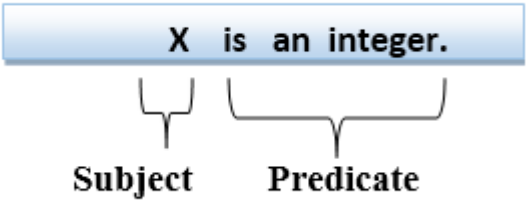
Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:
    a. **Universal Quantifier, (for all, everyone, everything)**
    b. **Existential quantifier, (for some, at least one).**

Universal Quantifier:

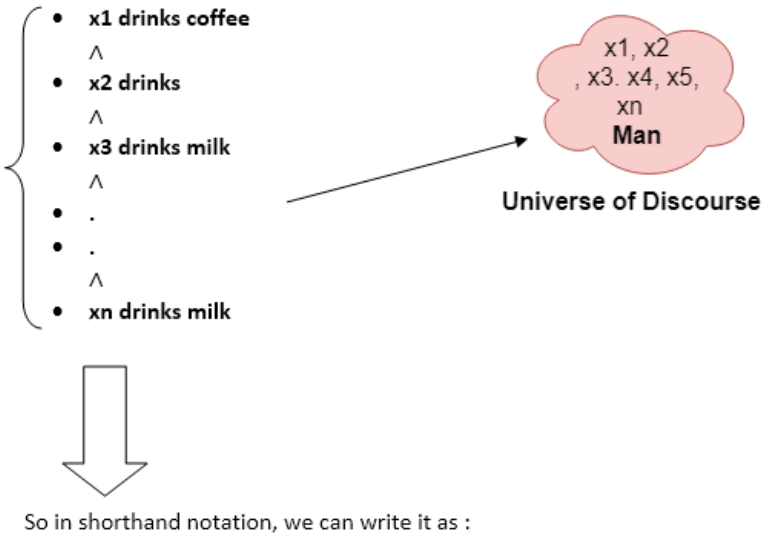
Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.
The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.
If x is a variable, then ∀x is read as

- **For all x**
- **For each x**
- **For every x.**

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:



∀x man(x) → drink (x, coffee).

It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifier:

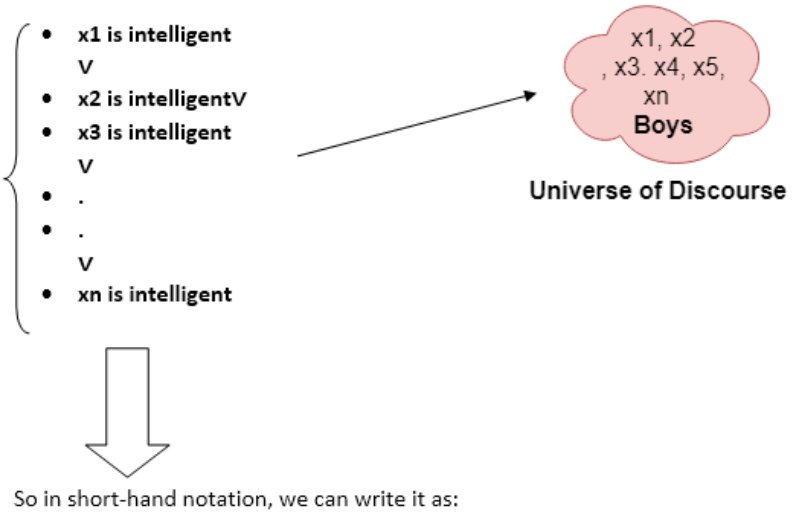
Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

- If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

  There exists a 'x.'
- **For some 'x.'**
- **For at least one 'x.'**

Example



∃x: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

- The main connective for universal quantifier ∀ is implication →.
- The main connective for existential quantifier ∃ is and ∧.

Some Examples of FOL using quantifier:

1. All birds fly.
In this question the predicate is "**fly(bird)**."
And since there are all birds who fly so it will be represented as follows.

∀x bird(x) →fly(x).

2. Every man respects his parent.
In this question, the predicate is "**respect(x, y)," where x=man, and y= parent**.Since there is every man so will use ∀, and it will be represented as follows:

∀x man(x) → respects (x, parent).

3. Some boys play cricket.
In this question, the predicate is "**play(x, y)**," where x= boys, and y= game. Since there are some boys so we will use **∃, and it will be represented as**:

∃x boys(x) → play(x, cricket).

4. Not all students like both Mathematics and Science.
In this question, the predicate is "**like(x, y)," where x= student, and y= subject**.
Since there are not all students, so we will use **∀ with negation, so** following representation for this:

¬∀ (x) [student(x) → like(x, Mathematics) ∧ like(x, Science)].

5.Only one student failed in Mathematics.

In this question, the predicate is "**failed(x, y),**" **where x= student, and y= subject**.

Since there is only one student who failed in Mathematics, so we will use following representation for this:

∃(x) [student(x) → failed (x, Mathematics) ∧∀ (y) [¬(x==y) ∧ student(y) → ¬failed (x, Mathematics)].

Representation Revisited

Programming languages (such as C++ or Java or Python) are the largest class of formal languages in common use. Data structures within programs can be used to represent facts; for example, a program could use a 4 ×4 array to represent the contents of the wumpus world. Thus, the programming language statement World[2,2] ←Pit is a fairly natural way to assert that there is a pit in square [2,2]. Putting together a string of such statements is sufficient for running a simulation of the wumpus world. What programming languages lack is a general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain. This procedural approach can be contrasted with the declarative nature of propositional logic, in which knowledge and inference are separate, and inference is entirely domain independent. SQL databases take a mix of declarative and procedural knowledge.

A second drawback of data structures in programs (and of databases) is the lack of any easy way to say, for example, “There is a pit in [2,2] or [3,1]” or “If the wumpus is in [1,1] then he is not in [2,2].” Programs can store a single value for each variable, and some systems allow the value to be unknown,” but they lack the expressiveness required to directly handle partial information.

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation. Propositional logic has a third property that is desirable in representation languages, namely, compositionality.

The language of thought Natural languages (such as English or Spanish) are very expressive indeed. We managed to write almost this whole book in natural language, with only occasional lapses into other languages (mainly mathematics

and diagrams). There is a long tradition in linguistics and the philosophy of language that views natural language as a declarative knowledge representation language. If we could uncover the rules for natural language, we could use them in representation and reasoning systems and gain the benefit of the billions of pages that have been written in natural language.

The modern view of natural language is that it serves as a medium for communication rather than pure representation. When a speaker points and says, "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence "Look!" represents that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the context in which the sentence was spoken. Clearly, one could not store a sentence such as "Look!" in a knowledge base and expect to recover its meaning without also storing a representation of the context—which raises the question of how the context itself can be represented.

Combining the best of formal and natural languages

We can adopt the foundation of propositional logic—a declarative, compositional semantics that is context-independent and unambiguous—and build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks.
When we look at the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to objects (squares, pits, wumpuses) and verbs and verb phrases along with adjectives and adverbs that refer to relations among objects (is breezy, is adjacent to, shoots). Some of these relations are functions—relations in which there is only one "value" for a given "input." It is easy to start listing examples of objects, relations, and functions:

• Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries . . .

• Relations: these can be unary relations or properties such as red, round, bogus, prime, multistoried . . ., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, . . .

• Functions: father of, best friend, third inning of, one more than, beginning of . . .
Indeed, almost any assertion can be thought of as referring to objects and properties or relations.

Some examples follow:

• “One plus two equals three.”
Objects: one, two, three, one plus two; Relation: equals; Function: plus. (“One plus two” is a name for the object that is obtained by applying the function “plus” to the objects “one” and “two.” “Three” is another name for this object.)

• “Squares neighboring the wumpus are smelly.”
Objects: wumpus, squares; Property: smelly; Relation: neighboring.

• “Evil King John ruled England in 1200.”
Objects: John, England, 1200; Relation: ruled during; Properties: evil, king.
The language of first-order logic, whose syntax and semantics we define in the next section, is built around objects and relations. It has been important to mathematics, philosophy, and artificial intelligence precisely because those fields—and indeed, much of everyday human existence—can be usefully thought of as dealing with objects and the relations among them.
First-order logic can also express facts about some or all of the objects in the universe. This enables one to represent general laws or rules, such as the statement “Squares neighboring the wumpus are smelly.”

Formal languages and their ontological and epistemological commitments.

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0,1]$
Fuzzy logic	facts with degree of truth $\in [0,1]$	known interval value

Using First-Order Logic

In knowledge representation a domain is just some part of the world about which we wish to express some knowledge.

We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of family relationships, numbers, sets, and lists, and at the wumpus world.

Assertions and queries in first-order logic

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such
sentences are called assertions. For example, we can assert that John is a king, Richard is a
person, and all kings are persons:
TELL(KB, King(John)) .
TELL(KB, Person(Richard)) .
TELL(KB, ∀x King(x) ⇒ Person(x)) .
We can ask questions of the knowledge base using ASK. For example,
ASK(KB, King(John))
returns true. Questions asked with ASK are called queries or goals. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the three assertions above, the query
ASK(KB, Person(John))
should also return true. We can ask quantified queries, such as
ASK(KB, ∃x Person(x)) .
The answer is true, but this is perhaps not as helpful as we would like. It is rather like answering "Can you tell me the time?" with "Yes." If we want to know what value of x makes the sentence true, we will need a different function, which we call ASKVARS,
ASK VARS(KB, Person(x))
and which yields a stream of answers. In this case there will be two answers: {x/John}
and {x/Richard}. Such an answer is called a substitution or binding list. ASKVARS is
usually reserved for knowledge bases consisting solely of Horn clauses, because in such
knowledge bases every way of making the query true will bind the variables to specific values.
That is not the case with first-order logic; in a KB that has been told only that King(John) ∨ King(Richard) there is no single binding to x that makes the query ∃x King(x) true, even though the query is in fact true.

Numbers, sets, and lists

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of natural numbers or nonnegative integers. We need a predicate NatNum that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S

(successor). The Peano axioms define natural numbers and addition. Natural numbers are defined recursively:

$$NatNum(0).$$
$$\forall n \; NatNum(n) \Rightarrow NatNum(S(n)).$$

That is, 0 is a natural number, and for every object n, if n is a natural number, then $S(n)$ is a natural number. So the natural numbers are 0, $S(0)$, $S(S(0))$, and so on. We also need axioms to constrain the successor function:

$$\forall n \; 0 \neq S(n).$$
$$\forall m,n \; m \neq n \Rightarrow S(m) \neq S(n).$$

Now we can define addition in terms of the successor function:

$$\forall m \; NatNum(m) \Rightarrow +(0,m) = m.$$
$$\forall m,n \; NatNum(m) \wedge NatNum(n) \Rightarrow +(S(m),n) = S(+(m,n)).$$

The first of these axioms says that adding 0 to any natural number m gives m itself. Notice the use of the binary function symbol "+" in the term $+(m,0)$; in ordinary mathematics, the term would be written $m+0$ using **infix** notation. (The notation we have used for first-order logic is called **prefix**.) To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write $S(n)$ as $n+1$, so the second axiom becomes

$$\forall m,n \; NatNum(m) \wedge NatNum(n) \Rightarrow (m+1)+n = (m+n)+1.$$

This axiom reduces addition to repeated application of the successor function.

The use of infix notation is an example of syntactic sugar, that is, an extension to or abbreviation of the standard syntax that does not change the semantics. Any sentence that uses sugar can be "desugared" to produce an equivalent sentence in ordinary first-order logic.

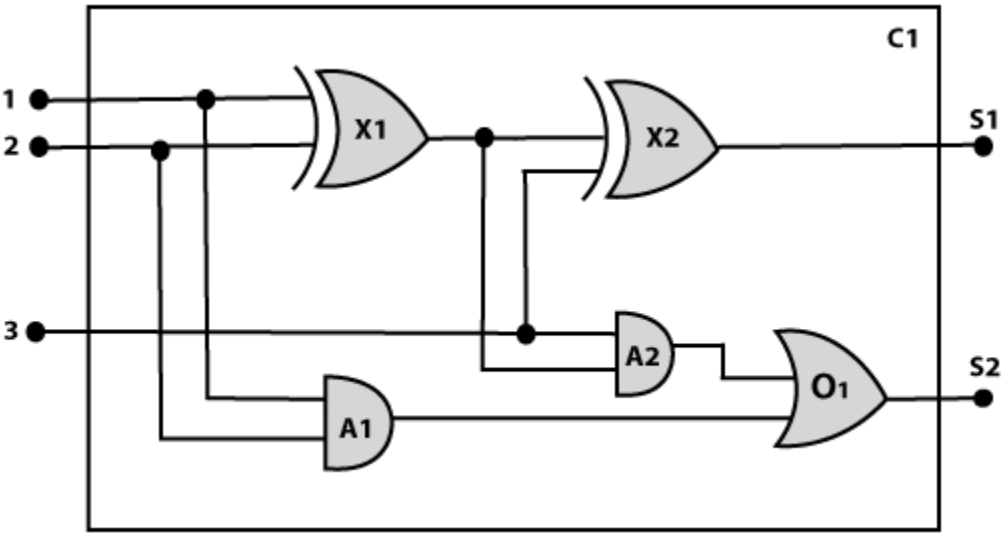
KNOWLEDGE ENGINEERING IN FIRST-ORDER LOGIC

What is knowledge-engineering?

The process of constructing a knowledge-base in first-order logic is called as knowledge- engineering. In **knowledge-engineering**, someone who investigates a particular domain, learns important concept of that domain, and generates a formal representation of the objects, is known as **knowledge engineer**.

The knowledge-engineering process:

Following are some main steps of the knowledge-engineering process. Using these steps, we will develop a knowledge base which will allow us to reason about digital circuit (**One-bit full adder**) which is given below.



1. Identify the task:

The first step of the process is to identify the task, and for the digital circuit, there are various reasoning tasks.

At the first level or highest level, we will examine the functionality of the circuit:

- **Does the circuit add properly?**
- **What will be the output of gate A2, if all the inputs are high?**

At the second level, we will examine the circuit structure details such as:

- **Which gate is connected to the first input terminal?**
- **Does the circuit have feedback loops?**

2. Assemble the relevant knowledge:

In the second step, we will assemble the relevant knowledge which is required for digital circuits. So for digital circuits, we have the following required knowledge:

- Logic circuits are made up of wires and gates.
- Signal flows through wires to the input terminal of the gate, and each gate produces the corresponding output which flows further.
- In this logic circuit, there are four types of gates used: **AND, OR, XOR, and NOT**.
- All these gates have one output terminal and two input terminals (except NOT gate, it has one input terminal).

3. Decide on vocabulary:

The next step of the process is to select functions, predicate, and constants to represent the circuits, terminals, signals, and gates. Firstly we will distinguish the gates from each other and from other objects. Each gate is represented as an object which is named by a constant, such as, **Gate(X1)**. The functionality of each gate is determined by its type, which is taken as constants such as **AND, OR, XOR, or NOT**. Circuits will be identified by a predicate: **Circuit (C1)**.
For the terminal, we will use predicate: **Terminal(x)**.

For gate input, we will use the function **In(1, X1)** for denoting the first input terminal of the gate, and for output terminal we will use **Out (1, X1)**.

The function **Arity(c, i, j)** is used to denote that circuit c has i input, j output.
The connectivity between gates can be represented by predicate
Connect (Out(1, X1), In(1, X1)).
We use a unary predicate **On (t)**, which is true if the signal at a terminal is on.

Encode general knowledge about the domain:

To encode the general knowledge about the logic circuit, we need some following rules:

- If two terminals are connected then they have the same input signal, it can be represented as:

∀ t1, t2 Terminal (t1) ∧ Terminal (t2) ∧ Connect (t1, t2) → Signal (t1) = Signal (2).

- Signal at every terminal will have either value 0 or 1, it will be represented as:
  ∀ t Terminal (t) →Signal (t) = 1 ∨Signal (t) = 0.

- Connect predicates are commutative:
  ∀ t1, t2 Connect(t1, t2) → Connect (t2, t1).

Representation of types of gates:

∀ g Gate(g) ∧ r = Type(g) → r = OR ∨r = AND ∨r = XOR ∨r = NOT.

- Output of AND gate will be zero if and only if any of its input is zero.

  ∀ g Gate(g) ∧ Type(g) = AND →Signal (Out(1, g))= 0 ⇔ ∃n Signal (In(n, g))= 0.

- Output of OR gate is 1 if and only if any of its input is 1:

  ∀ g Gate(g) ∧ Type(g) = OR → Signal (Out(1, g))= 1 ⇔ ∃n Signal (In(n, g))= 1

- Output of XOR gate is 1 if and only if its inputs are different:

  ∀ g Gate(g) ∧ Type(g) = XOR → Signal (Out(1, g)) = 1 ⇔ Signal (In(1, g)) ≠ Signal (In(2, g)).

- Output of NOT gate is invert of its input:

  ∀ g Gate(g) ∧ Type(g) = NOT → Signal (In(1, g)) ≠ Signal (Out(1, g)).
  All the gates in the above circuit have two inputs and one output (except NOT gate).

  ∀ g Gate(g) ∧ Type(g) = NOT → Arity(g, 1, 1)

∀ g Gate(g) ∧ r =Type(g) ∧ (r= AND ∨r= OR ∨r= XOR) → Arity (g, 2, 1).

- All gates are logic circuits:

  ∀ g Gate(g) → Circuit (g).

Encode a description of the problem instance:

Now we encode problem of circuit C1, firstly we categorize the circuit and its gate components. This step is easy if ontology about the problem is already thought. This step involves the writing simple atomics sentences of instances of concepts, which is known as ontology.

For the given circuit C1, we can encode the problem instance in atomic sentences as below:

Since in the circuit there are two XOR, two AND, and one OR gate so atomic sentences for these gates will be:

1. For XOR gate: Type(x1)= XOR, Type(X2) = XOR
2. For AND gate: Type(A1) = AND, Type(A2)= AND
3. For OR gate: Type (O1) = OR.

Pose queries to the inference procedure and get answers:

In this step, we will find all the possible set of values of all the terminal for the adder circuit. The first query will be:

What should be the combination of input which would generate the first output of circuit C1, as 0 and a second output to be 1?

$\exists$ i1, i2, i3 Signal (In(1, C1))=i1 $\wedge$ Signal (In(2, C1))=i2 $\wedge$ Signal (In(3, C1))= i3
$\wedge$ Signal (Out(1, C1)) =0 $\wedge$ Signal (Out(2, C1))=1

Debug the knowledge base:

Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of knowledge base.

In the knowledge base, we may have omitted assertions like $1 \neq 0$.

INFERENCE IN FIRST-ORDER LOGIC

Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences. Before understanding the FOL inference rule, let's understand some basic terminologies used in FOL.

Substitution:

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write **F[a/x]**, so it refers to substitute a constant "**a**" in place of variable "**x**".

First-Order logic does not only use predicate and terms for making atomic sentences but also uses another way, which is equality in FOL. For this, we can use **equality symbols** which specify that the two terms refer to the same object.

Example: Brother (John) = Smith.

As in the above example, the object referred by the **Brother (John)** is similar to the object referred by **Smith**. The equality symbol can also be used with negation to represent that two terms are not the same objects.

Example: ¬(x=y) which is equivalent to x ≠y.

FOL inference rules for quantifier:

As propositional logic we also have inference rules in first-order logic, so following are some basic inference rules in FOL:

- **Universal Generalization**
- **Universal Instantiation**
- **Existential Instantiation**
- **Existential introduction**

1. Universal Generalization:

Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).

It can be represented as: $\frac{P(c)}{\forall x\, P(x)}$.

This rule can be used if we want to show that every element has a similar property.

In this rule, x must not appear as a free variable.

Example: Let's represent, P(c): "**A byte contains 8 bits**", so for **∀ x P(x)** "**All bytes contain 8 bits**.", it will also be true.

Universal Instantiation:

- Universal instantiation is also called as universal elimination or UI is a valid inference rule. It can be applied multiple times to add new sentences.
- The new KB is logically equivalent to the previous KB.
- As per UI, **we can infer any sentence obtained by substituting a ground term for the variable**.
- The UI rule state that we can infer any sentence P(c) by substituting a ground term c (a constant within domain x) from **∀ x P(x) for any object in the universe of discourse**.
- It can be represented as: $\frac{\forall x\, P(x)}{P(c)}$.

Example:1.

IF "Every person like ice-cream"=> ∀x P(x) so we can infer that "John likes ice-cream" => P(c)

Example: 2.

Let's take a famous example,
"All kings who are greedy are Evil." So let our knowledge base contains this detail as in the form of FOL:
∀x king(x) ∧ greedy (x) → Evil (x),
So from this information, we can infer any of the following statements using

Universal Instantiation:

- **King(John) ∧ Greedy (John) → Evil (John),**
- **King(Richard) ∧ Greedy (Richard) → Evil (Richard),**
- **King(Father(John)) ∧ Greedy (Father(John)) → Evil (Father(John)),**

Existential Instantiation:

- Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.
- It can be applied only once to replace the existential sentence.
- The new KB is not logically equivalent to old KB, but it will be satisfiable if old KB was satisfiable.
- This rule states that one can infer P(c) from the formula given in the form of ∃x P(x) for a new constant symbol c.
- The restriction with this rule is that c used in the rule must be a new term for which P(c) is true.

- It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Existential introduction

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as: $\frac{P(c)}{\exists x P(x)}$
- **Example: Let's say that,**

  "Priyanka got good marks in English."
  "Therefore, someone got good marks in English."

Generalized Modus Ponens Rule:
For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens.

Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."

According to Modus Ponens, for atomic sentences **pi, pi', q**. Where there is a substitution θ such that SUBST **(θ, pi',) = SUBST(θ, pi)**, it can be represented as:

$$\frac{p1', p2', \ldots, pn', (p1 \wedge p2 \wedge \ldots \wedge pn \Rightarrow q)}{SUBST(\theta, q)}$$

Example:

We will use this rule for Kings are evil, so we will find some x such that x is king, and x is greedy so we can infer that x is evil.

Here let say, p1' is king(John) p1 is king(x)
p2' is Greedy(y) p2 is Greedy(x)
θ is {x/John, y/John} q is evil(x)
SUBST(θ,q).

Propositional vs. First-Order Inference

Propositional Logic:

Propositional logic, also known as sentential logic, deals with propositions or statements that are either true or false. It focuses on the relationships between propositions using logical connectives (such as AND, OR, NOT, IMPLIES) and truth-functional operators. However, it doesn't deal with the internal structure of propositions or the quantification over variables.

Example propositions:

P: It is raining.

Q: The ground is wet.

Example expressions in propositional logic:

P AND Q (It is raining and the ground is wet.)

NOT P (It is not raining.)

P IMPLIES Q (If it is raining, then the ground is wet.)

First-Order Logic:

First-order logic (FOL), also known as first-order predicate logic or first-order predicate calculus, is a more expressive logic that allows for quantification over variables, relationships between objects, and internal structure within propositions. It includes predicates (relations), functions, quantifiers (such as ∀ for "for all" and ∃ for "exists"), and variables.

Example predicates and quantified expressions in first-order logic:

Loves(x, y) (Person x loves person y.)

∀x ∃y Loves(x, y) (Everyone loves someone.)

Key Differences:

Expressiveness: Propositional logic deals with simple true/false propositions and their combinations using logical operators. First-order logic goes beyond this by allowing the representation of complex relationships, quantification over variables, and functions that operate on objects.

Quantification: Propositional logic lacks quantifiers (like "for all" and "exists") which are essential for expressing general statements and relationships involving variables. First-order logic includes quantifiers to make statements about entire classes of objects.

Structure: In propositional logic, propositions are atomic and not further decomposed. In first-order logic, propositions can contain variables, predicates, and functions, allowing for more detailed representation of relationships and properties.

Scope: Propositional logic is often used for simple reasoning tasks and truth tables. First-order logic is more suitable for representing complex relationships, making inferences, and expressing higher-level concepts.

In total, propositional logic deals with true/false propositions and their logical relationships, while first-order logic extends this by allowing quantification, variable binding, and representation of more intricate relationships between objects.

UNIFICATION IN FIRST-ORDER LOGIC.

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\boldsymbol{\Psi_1\sigma = \Psi_2\sigma}$, then it can be expressed as **UNIFY(Ψ_1, Ψ_2)**.
- **Example: Find the MGU for Unify{King(x), King(John)}**

Let Ψ_1 = King(x), Ψ_2 = King(John),

Substitution θ = {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).
- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

E.g. Let's say there are two different expressions, **P(x, y), and P(a, f(z))**.

In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

P(x,y).....(i)
P(a, f(z))..... (ii)

- Substitute x with a, and y with f(z) in the first expression, and it will be represented as **a/x** and f(z)/y.
- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: **[a/x, f(z)/y]**.

Conditions for Unification:

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm:

Algorithm: Unify(Ψ_1, Ψ_2)

Step. 1: If Ψ_1 or Ψ_2 is a variable or constant, then:
a) If Ψ_1 or Ψ_2 are identical, then return NIL.
b) Else if $\Psi_1$is a variable,
a. then if Ψ_1 occurs in Ψ_2, then return FAILURE
b. Else return { (Ψ_2/ Ψ_1)}.
c) Else if Ψ_2 is a variable,
a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
b. Else return {(Ψ_1/ Ψ_2)}.
d) Else return FAILURE.
Step.2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.
Step. 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.
Step. 4: Set Substitution set(SUBST) to NIL.
Step. 5: For i=1 to the number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

Implementation of the Algorithm

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

Step.2: Recursively unify atomic sentences:

a. Check for Identical expression match.

b. If one expression is a variable v_i, and the other is a term t_i which does not contain variable v_i, then:

a. Substitute t_i / v_i in the existing substitutions

b. Add t_i /v_i to the substitution setlist.

c. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

For each pair of the following atomic sentences find the most general unifier (If exist).

1. Find the MGU of {p(f(a), g(Y)) and p(X, X)}

Sol: S0 => Here, Ψ1 = p(f(a), g(Y)), and Ψ2 = p(X, X)

SUBST θ= {f(a) / X}

S1 => Ψ1 = p(f(a), g(Y)), and Ψ2 = p(f(a), f(a))

SUBST θ= {f(a) / g(y)}, Unification failed.

Unification is not possible for these expressions.

2. Find the MGU of {p(b, X, f(g(Z))) and p(Z, f(Y), f(Y))}

Here, Ψ1 = p(b, X, f(g(Z))) , and Ψ2 = p(Z, f(Y), f(Y))

S0 => { p(b, X, f(g(Z))); p(Z, f(Y), f(Y))}

SUBST θ={b/Z}

S1 => { p(b, X, f(g(b))); p(b, f(Y), f(Y))}

SUBST θ={f(Y) /X}

S2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))}

SUBST θ= {g(b) /Y}

S2 => { p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))} Unified Successfully.

And Unifier = { b/Z, f(Y) /X , g(b) /Y}.

3. Find the MGU of {p (X, X), and p (Z, f(Z))}

Here, Ψ1 = {p (X, X), and Ψ2 = p (Z, f(Z))

S0 => {p (X, X), p (Z, f(Z))}

SUBST θ= {X/Z}

S1 => {p (Z, Z), p (Z, f(Z))}

SUBST θ= {f(Z) / Z}, Unification Failed.

Hence, unification is not possible for these expressions.

4. Find the MGU of UNIFY(prime (11), prime(y))

Here, Ψ1 = {prime(11) , and Ψ2 = prime(y)}

S0 => {prime(11) , prime(y)}

SUBST θ= {11/y}

S1 => {prime(11) , prime(11)} , Successfully unified.

Unifier: {11/y}.

5. Find the MGU of Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)}

Here, Ψ1 = Q(a, g(x, a), f(y)), and Ψ2 = Q(a, g(f(b), a), x)

S0 => {Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)}

SUBST θ= {f(b)/x}

S1 => {Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))}

SUBST θ= {b/y}

S1 => {Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))}, Successfully Unified.

Unifier: [a/a, f(b)/x, b/y].

6. UNIFY(knows(Richard, x), knows(Richard, John))

Here, Ψ1 = knows(Richard, x), and Ψ2 = knows(Richard, John)

S0 => { knows(Richard, x); knows(Richard, John)}

SUBST θ= {John/x}

S1 => { knows(Richard, John); knows(Richard, John)}, Successfully Unified.

Unifier: {John/x}.

Forward Chaining

First-order definite clauses

First-order definite clauses are disjunctions of literals of which exactly one is positive. That means a definite clause is either atomic, or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. Existential quantifiers are not allowed, and universal quantifiers are left implicit: if you see an x in a definite clause, that means there is an implicit ∀x quantifier. A typical first-order definite clause looks like this:

King(x) ∧Greedy(x) ⇒ Evil(x) ,

but the literals King(John) and Greedy(y) also count as definite clauses. First-order literals can include variables, so Greedy(y) is interpreted as “everyone is greedy” (the universal quantifier is implicit).
Let us put definite clauses to work in representing the following problem:

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

First, we will represent these facts as first-order definite clauses:
“. . . it is a crime for an American to sell weapons to hostile nations”:

American(x) ∧Weapon(y) ∧Sells(x, y, z) ∧Hostile(z) ⇒ Criminal(x) . ---------- (1)
"Nono . . . has some missiles." The sentence ∃x Owns(Nono, x) ∧ Missile(x) is transformed
into two definite clauses by Existential Instantiation, introducing a new constant M_1:
Owns(Nono, M_1) ---------- (2)
Missile(M_1) ---------- (3)
"All of its missiles were sold to it by Colonel West":
Missile(x) ∧Owns(Nono, x) ⇒ Sells(West, x, Nono) ---------- 4)

We will also need to know that missiles are weapons:
Missile(x) ⇒Weapon(x) ---------- (5)
and we must know that an enemy of America counts as "hostile":
Enemy(x, America) ⇒ Hostile(x) . ---------- (6)
"West, who is American . . .":
American(West) . ---------- (7)
"The country Nono, an enemy of America . . .":
Enemy(Nono, America) . ---------- (8)

This knowledge base happens to be a Datalog knowledge base: Datalog is a language consisting of first-order definite clauses with no function symbols. Datalog gets its name because
it can represent the type of statements typically made in relational databases. The absence of
function symbols makes inference much easier.

A simple forward-chaining algorithm

A simple forward chaining inference algorithm is shown below

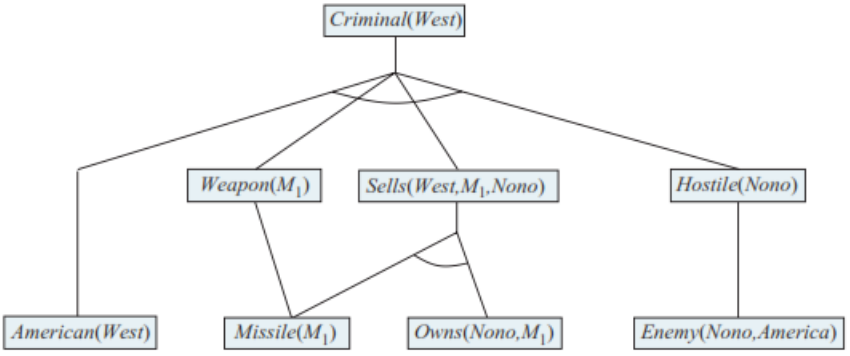
```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence

  while true do
    new ← { }        // The set of new sentences inferred on each iteration
    for each rule in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new then
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not failure then return φ
    if new = { } then return false
    add new to KB
```

Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered (assuming that just one answer is required) or no new

facts are added. Notice that a fact is not "new" if it is just a renaming of a known fact—a sentence is a renaming of another if they are identical except for the names of the variables. For example, Likes(x, IceCream) and Likes(y, IceCream) are renamings of each other. They both mean the same thing: "Everyone likes ice cream."
We use our crime problem to illustrate FOL-FC-ASK. The implication sentences available for chaining are (1), (4), (5), and (6). Two iterations are required:

• On the first iteration, rule (9.3) has unsatisfied premises.
Rule (4) is satisfied with $\{x/M_1\}$, and Sells(West, M_1, Nono) is added.
Rule (5) is satisfied with $\{x/M_1\}$, and Weapon(M_1) is added.
Rule (6) is satisfied with {x/Nono}, and Hostile(Nono) is added.

• On the second iteration, rule (1) is satisfied with $\{x/West, y/M_1, z/Nono\}$, and the
inference Criminal(West) is added.



The above figure shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a fixed point of the inference process. Fixed points reached by forward chaining with first-order definite clauses are similar to those for propositional forward chaining ; the principal difference is that a first order fixed point can include universally quantified atomic sentences.
FOL-FC-ASK is easy to analyze. First, it is sound, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is complete for definite
clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses.
For Datalog knowledge bases, which contain no function symbols, the proof of completeness is fairly easy. We begin by counting the number of possible facts that can be added, which determines the maximum number of iterations. Let k be the maximum arity (number of arguments) of any predicate, p be the number of predicates, and n be the number of constant symbols. Clearly, there can be no

more than pn^k distinct ground facts, so after this many iterations the algorithm must have reached a fixed point.

For general definite clauses with function symbols, FOL-FC-ASK can generate infinitely many new facts, so we need to be more careful. For the case in which an answer to the query sentence q is entailed, we must appeal to Herbrand's theorem to establish that the algorithm will find a proof. If the query has no answer, the algorithm could fail to terminate in some cases. For example, if the knowledge base includes the Peano axioms
NatNum(0)
$\forall n$ NatNum(n) $\Rightarrow$ NatNum(S(n)) ,
then forward chaining adds NatNum(S(0)), NatNum(S(S(0))), NatNum(S(S(S(0)))), and so on. This problem is unavoidable in general. As with general first-order logic, entailment with definite clauses is semidecidable.

BACKWARD CHAINING

The second major family of logical inference algorithms uses the **backward chaining** approach for definite clauses. These algorithms work backward from the goal, chaining through rules to find known facts that Support the proof.

A backward-chaining algorithm

Figure below shows a backward-chaining algorithm for definite clauses. FOL-BC- ASK(KB, goal) will be proved if the knowledge base contains a clause of the form lhs $\Rightarrow$ goal, where lhs (left-hand side) is a list of conjuncts. An atomic fact like American(West) is considered as a clause whose lhs is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query Person(x) could be proved with the substitution {x/John} as well as with {x/Richard }. So we implement FOL-BC-ASK
as a **generator**— a function that returns multiple times, each time giving one possible result.

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

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

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

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

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the rhs of the clause does indeed unify with the goal, proving every conjunct in the lhs, using FOL-BC-AND. That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go. Figure 9.7 is the proof tree for deriving Criminal (West) from sentences (1) through (8).

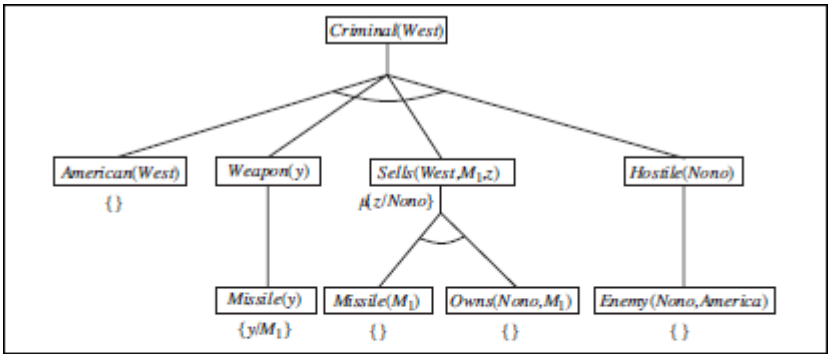


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal*(*West*), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally *Hostile*(*z*), *z* is already bound to *Nono*.

Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the

proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness