UNIT IV LOGICAL AGENTS

Knowledge-based agents –propositional logic –propositional theorem proving –propositional model checking –agents based on propositional logic
First-order logic –syntax and semantics –knowledge representation and engineering – inferences in first-order logic –forward chaining –backward chaining –resolution

KNOWLEDGE-BASED AGENTS

The central component of a knowledge-based agent is its **knowledge base**, or KB. A knowledge base is a set of **sentences**. (Here "sentence" is used as a technical term. It is related but not identical to the sentences of English and other natural languages.) Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. When the sentence is taken as being given without being derived from other sentences, it is called as an **axiom**.

There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively. Both operations may involve **inference**—that is, deriving new sentences from old. Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously.

Figure shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, , which may initially contain some **background knowledge**.

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

   TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
   action ← ASK(KB, MAKE-ACTION-QUERY(t))
   TELL(KB, MAKE-ACTION-SENTENCE(action, t))
   t ← t + 1
   return action
```

Each time the agent program is called, it does **three things**. First, it TELLs the knowledge base what it perceives. Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLs the knowledge base which action was chosen, and returns the action so that it can be executed.

The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other. MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time. MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time. Finally, MAKE ACTIONSENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK.

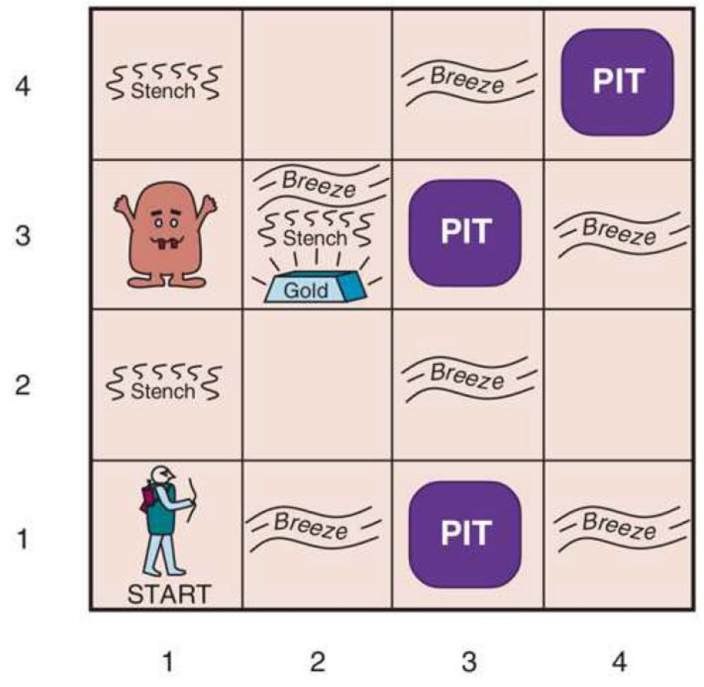
Because of the definitions of TELL and ASK, however, the knowledge-based agent is not an arbitrary program for calculating actions. It is a description at the **knowledge level**, where the description and goals known to agent are specified in order to determine its behavior.

A knowledge-based agent can be built simply by TELLing it what it needs to know. Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment. This is called the **declarative** approach to system building. In contrast, the **procedural** approach encodes desired behaviors directly as program code.

The Wumpus World

The **wumpus world** is a cave consisting of rooms connected by passageways, somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms. The only redeeming feature of this bleak environment is the possibility of finding a heap of gold.

A sample Wumpus world is shown in **Figure**. The agent is in the bottom left corner, facing east (rightward).



The precise definition of the task environment is given by the PEAS description

PERFORMANCE MEASURE: +1000 for climbing out of the cave with the gold, -1000 for falling into a pit or being eaten by the wumpus, -1 for each action taken, and -10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
ENVIRONMENT: A 4x4 grid of rooms, with walls surrounding the grid. The agent always starts in the square labeled [1,1], facing to the east. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
ACTUATORS: The agent can move *Forward*, *TurnLeft* by 90°, or *TurnRight* by 90°. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. If an agent tries to move forward and bumps into a wall, then the agent does not move. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect. Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].
SENSORS: The agent has five sensors, each of which gives a single bit of information:
- In the squares directly (not diagonally) adjacent to the wumpus, the agent will perceive a *Stench*.
- In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
- In the square where the gold is, the agent will perceive a *Glitter*.
- When an agent walks into a wall, it will perceive a *Bump*.
- When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [*Stench*,*Breeze*,*None*,*None*,*None*].

The Wumpus world Properties:
Partially observable: The Wumpus world is partially observable because the agent can only perceive the close environment such as an adjacent room.
Deterministic: It is deterministic, as the result and outcome of the world are already known.
Sequential: The order is important, so it is sequential.
Static: It is static as Wumpus and Pits are not moving.
Discrete: The environment is discrete.
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
The agent's initial knowledge base contains the rules of the environment and it knows that the agent is in [1,1] it is a safe square; This is denoted with an "A" and "OK," respectively, in square [1,1].

The first percept is [*None*,*None*,*None*,*None*,*None*], from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK. Figure (a) shows the agent's state of knowledge at this point.

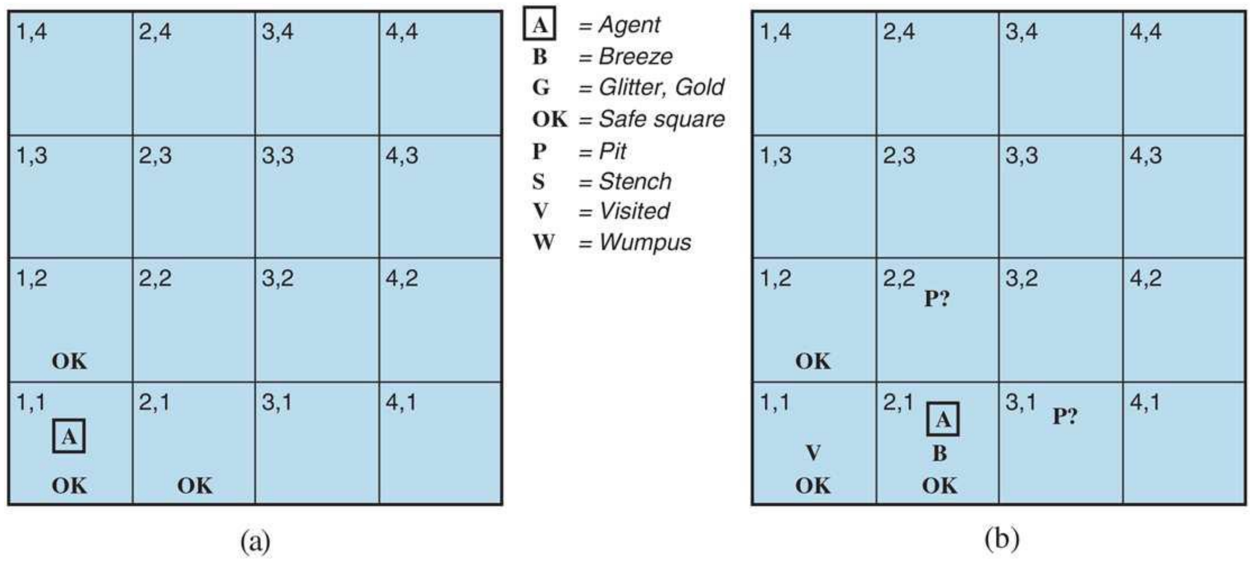


Figure: The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None, None, None, None, None*].(b) After moving to [2,1] and perceiving [*None, Breeze, None, None, None*].

A cautious agent will move only into a square that it knows to be OK. Suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by "B") in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation "P?" in Figure (b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

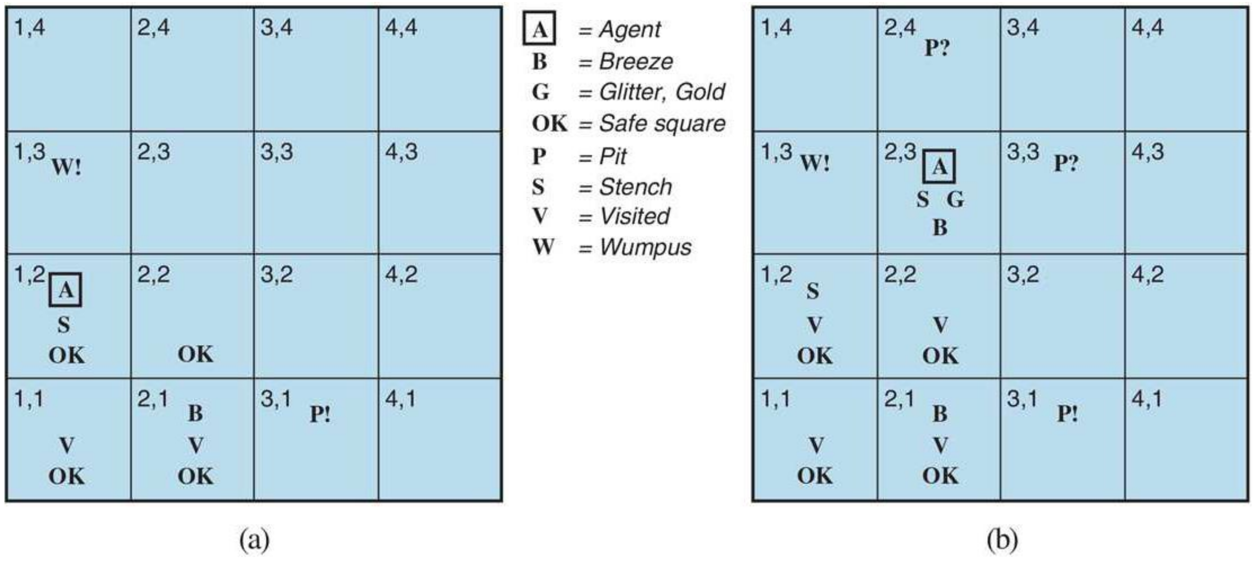


Figure: Two later stages in the progress of the agent. (a) After moving to [1,1] and then [1,2], and perceiving [*Stench*,*None*,*None*,*None*,*None*]. (b) After moving to [2,2] and then [2,3], and perceiving[Stench,Breeze,Glitter,None,None].

The agent perceives a stench in [1,2], resulting in the state of knowledge shown in Figure (a).The stench in [1,2] means that there must be a wumpus nearby. But the Wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the Wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. Assume that the agent turns and moves to [2,3], as in Figure (b) . In [2,3], the agent detects a glitter, so it should grab the gold and then return home. Note that in each case for which the agent draws a conclusion from the available information, that conclusion is *guaranteed* to be correct if the available information is correct. This is a fundamental property of logical reasoning.

PROPOSITIONAL LOGIC

Knowledge base consist of sentences. Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. Sometimes a sentence can be named as an **axiom**, when the sentence is taken as given without being derived from other sentences. These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed

- A knowledge representation language in which syntax and semantics are defined correctly is known as logic. The notion of syntax is clear enough in ordinary arithmetic:
  "x + y = 4" is a well-formed sentence, whereas "x4y+ =" is not.

A logic must also define the **semantics** or meaning of sentences. The semantics defines the **truth** of each sentence with respect to each **possible world**. For example, the semantics for arithmetic specifies that the sentence "x + y =4" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no "in between."1

Models are mathematical abstractions, each of which simply fixes model m, If a sentence α is true in model then m **satisfies** α or sometimes m **is a model of** α. The notation M(α) is used to mean the set of all models of α.The notion of truth involves the relation of logical **entailment** between sentences—the idea that a sentence *follows logically* from another sentence. In mathematical notation, this can be represented as **α |= β**

Syntax:

The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false.

Complex sentences are constructed from simpler sentences, using parentheses and **logical connectives**. There are five connectives in common use are:

- NEGATION ¬ (not). A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).

- CONJUNCTION∧ (and).
- DISJUNCTION ∨ (or)
- IMPLICATION ⇒ (implies). (or conditional). The initial statements are **premise** or **antecedent** , and its **conclusion** or **consequent**. Implications are also known as **rules** or**if–then** statements. The implication symbol is sometimes written in other books as ⊃ or →.
- BICONDITIONAL ⇔ (if and only if).**biconditional**. Some other books write this as ≡.

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

Sentence → AtomicSentence | ComplexSentence
AtomicSentence → True | False | P | Q | R | . . .
ComplexSentence → (Sentence)

| ¬ Sentence

| Sentence ∧ Sentence
| Sentence ∨ Sentence
| Sentence ⇒ Sentence
| Sentence ⇔ Sentence

OPERATOR PRECEDENCE : ¬ , ∧, ∨,⇒,⇔

Semantics
In propositional logic, a model simply fixes the **truth value**—true or false—for every proposition symbol.
The semantics for propositional logic must specify how to compute the truth value of *any* sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives.
Atomic sentences are easy:
• True is true in every model and False is false in every model.
• The truth value of every other proposition symbol must be specified directly in the model
For complex sentences,

• ¬ P is true iff P is false in m.

• P ∧ Q is true iff both P and Q are true in m.
• P ∨ Q is true iff either P or Q is true in m.
• P ⇒ Q is true unless P is true and Q is false in m.
• P ⇔ Q is true iff P and Q are both true or both false in m.
The rules can also be expressed with **truth tables** that specify the truth value of a complex sentence for each possible assignment of truth values to its components

The following truth table shows the five logical connectives

P	Q	¬ P	P Λ Q	P V Q	P ⇒ Q	P ⇔ Q
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

A simple inference procedure algorithm for inference KB |= α for some sentence α. is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that α is true in every model in which KB is true. Models are assignments of true or false to every proposition symbol.

A general algorithm for deciding entailment in propositional logic is shown in Figure .

```
function TT-ENTAILS?(KB,α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositional logic
symbols ←a list of the proposition symbols in KB and α
return TT-CHECK-ALL(KB,α, symbols, { })
function TT-CHECK-ALL(KB,α, symbols ,model ) returns true or false
if EMPTY?(symbols) then
if PL-TRUE?(KB,model ) then return PL-TRUE?(α,model )
else return true // when KB is false, always return true
else
P ←FIRST(symbols)
rest ←REST(symbols)
return (TT-CHECK-ALL(KB,α, rest ,model ∪ {P = true})
and
TT-CHECK-ALL(KB,α, rest ,model ∪ {P = false }))
```

The algorithm is **sound** because it implements directly the definition of entailment, and **complete** because it works for any KB and α and always terminates—there are only finitely many models to examine. Of course, "finitely many" is not always the same as "few." If KB and α contain n symbols in all, then there are 2^n models. Thus, the time complexity of the algorithm is $O(2^n)$.

PROPOSITIONAL THEOREM PROVING

Entailment can be done by **theorem proving**—applying rules of inference directly to the sentences in the knowledge base to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking.

The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. This can be written as α ≡ β.

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

1)(α ∧ β) ≡ (β ∧ α) commutativity of ∧
2)(α ∨ β) ≡ (β ∨ α) commutativity of ∨
3)((α ∧ β) ∧ γ) ≡ (α ∧ (β ∧ γ)) associativity of ∧

4)$((\alpha \lor \beta) \lor \gamma) \equiv (\alpha \lor (\beta \lor \gamma))$ associativity of $\lor$
5)$\neg(\neg\alpha) \equiv \alpha$ double-negation elimination
6)$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$ contraposition
7)$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \lor \beta)$ implication elimination
8)$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha))$ biconditional elimination
9)$\neg(\alpha \land \beta) \equiv (\neg\alpha \lor \neg\beta)$ De Morgan
10)$\neg(\alpha \lor \beta) \equiv (\neg\alpha \land \neg\beta)$ De Morgan
11)$(\alpha \land (\beta \lor \gamma)) \equiv ((\alpha \land \beta) \lor (\alpha \land \gamma))$ distributivity of $\land$ over $\lor$
12)$(\alpha \lor (\beta \land \gamma)) \equiv ((\alpha \lor \beta) \land (\alpha \lor \gamma))$ distributivity of $\lor$ over $\land$

Equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other: $\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$.

The second concept is **validity**. A sentence is valid if it is true in *all* models. For example, the sentence $P \lor \neg P$ is valid. Valid sentences are also known as **tautologies**—they are *necessarily* true. Because the sentence is true in all models, every valid sentence is logically equivalent to True.

DEDUCTION THEOREM
For any sentences α *and* β, $\alpha \models \beta$ *if and only if the sentence* $(\alpha \Rightarrow \beta)$ *is valid.*

The final concept is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, *some* model. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence.
SAT in propositional logic—the **SAT** problem—was the first problem proved to be NP-complete. Many problems in computer science are really satisfiability problems. Validity and satisfiability are of course connected: α is valid iff $\neg\alpha$ is unsatisfiable; contrapositively, α is satisfiable iff $\neg\alpha$ is not valid.

$\alpha \models \beta$ *if and only if the sentence* $(\alpha \land \neg\beta)$ *is unsatisfiable.*

Proving β from α by checking the unsatisfiability of $(\alpha \land \neg\beta)$ corresponds exactly to the standard mathematical proof technique of *reductio ad absurdum* It is also called proof by **refutation** or proof by **contradiction**. One assumes a sentence β to be false and shows that this leads to a contradiction with known axioms α. This contradiction is exactly what is meant by saying that the sentence $(\alpha \land \neg\beta)$ is unsatisfiable.

Inference and proofs
Inference rules can be applied to derive a **proof**—a chain of conclusions that leads to the desired goal. The best-known rule is called **Modus Ponens**

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}.$$

The notation means that, whenever any sentences of the form α ⇒ β and α are given, then the sentence β can be inferred. For example, if (WumpusAhead ∧WumpusAlive) ⇒ Shoot and (WumpusAhead ∧ WumpusAlive) are given, then Shoot can be inferred.

Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}.$$

For example, from (WumpusAhead ∧ WumpusAlive), WumpusAlive can be inferred. By considering the possible truth values of α and β, one can show easily that Modus Ponens and And-Elimination are sound once and for all. These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models.

For example, the equivalence for biconditional elimination yields the two inference rules

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

Example: Consider knowledge base containing rules R1 through R5

R1 : ¬ P1,1 .

R2 : B1,1 ⇔ (P1,2 ∨ P2,1) .
R3 : B2,1 ⇔ (P1,1 ∨ P2,2 ∨ P3,1) .

R4 : ¬ B1,1 .

R5 : B2,1 .

and show how to prove ¬ P1,2,

that is, there is no pit in [1,2]. First, apply biconditional elimination to R2 to obtain
R6: (B1,1 ⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .
Then apply And-Elimination to R6 to obtain
R7: ((P1,2 ∨ P2,1) ⇒ B1,1) .
Logical equivalence for contrapositives gives

R8: (¬ B1,1 ⇒ ¬ (P1,2 ∨ P2,1)) .

Now apply Modus Ponens with R8 and the percept R4 (i.e., ¬ B1,1), to obtain

R9 : ¬ (P1,2 ∨ P2,1) .

Finally, apply De Morgan's rule, giving the conclusion

R10 : ¬ P1,2 ∧ ¬ P2,1 .

That is, neither [1,2] nor [2,1] contains a pit.

A proof problem can be defined as follows:

• INITIAL STATE: the initial knowledge base.
• ACTIONS: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
• RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
• GOAL: the goal is a state that contains the sentence that has to be proved.

The final property of logical systems is **monotonicity**, which says that the set of entailed sentences can only *increase* as information is added to the knowledge base. For any sentences α and β, if KB |= α then KB ∧ β |= α .

Monotonicity means that inference rules can be applied whenever suitable premises are found in the knowledge base—the conclusion of the rule must follow *regardless of what else is in the knowledge base*.

Proof by resolution

A single inference rule, **resolution**, can yield a complete inference algorithm when coupled with any complete search algorithm.

Example :A simple version of the resolution rule in the wumpus world. Consider the agent returns from [2,1] to [1,1] and then goes to [1,2], where it perceives a stench, but no breeze. the following facts are added to the knowledge base:

R11 : ¬ B1,2 .

R12 : B1,2 ⇔ (P1,1 ∨ P2,2 ∨ P1,3) .

The absence of pits in [2,2] and [1,3] can be given as:

R13 : ¬ P2,2 .

R14 : ¬ P1,3 .

Biconditional elimination is applied to R3, followed by Modus Ponens with R5,
R3 : B2,1 ⇔ (P1,1 ∨ P2,2 ∨ P3,1) .
R5 : B2,1 .

to obtain the fact(Rule R15) that there is a pit in [1,1], [2,2], or [3,1]:
R15 : P1,1 ∨ P2,2 ∨ P3,1 .

Now comes the first application of the resolution rule: the literal ¬ P2,2 in R13 *resolves with* the literal P2,2 in R15 to give the **resolvent**
R16 : P1,1 ∨ P3,1 .

In English; if there’s a pit in one of [1,1], [2,2], and [3,1] and it’s not in [2,2], then it’s in [1,1] or [3,1]. Similarly, the literal ¬ P1,1 in R1 resolves with the literal P1,1 in R16 to give

R17 : P3,1 .

In English: if there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1]. These last two inference steps are examples of the **unit resolution** inference rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k},$$

where each l is a literal and li and m are **complementary literals** (i.e., one is the negation of the other). Thus, the unit resolution rule takes a **clause**—a disjunction of literals—and a literal and produces a new clause. Note that a single literal can be viewed as a disjunction of one literal, also known as a **unit clause**.

The unit resolution rule can be generalized to the full **resolution** rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n},$$

where li and mj are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses *except* the two complementary literals. For example,

$$\frac{P_{1,1} \vee P_{3,1}, \qquad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}.$$

There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal. The removal of multiple copies of literals is called **factoring**. For example, if (A ∨ B) is resolved with (A∨ ¬ B), (A ∨ A) is obtained, which is reduced to just A.

A simple resolution algorithm for propositional logic.

```
function PL-RESOLUTION(KB,α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositional logic
clauses ←the set of clauses in the CNF representation of KB ∧ ¬ α
new ←{}
loop do
for each pair of clauses Ci, Cj in clauses do
resolvents ←PL-RESOLVE(Ci,Cj )
if resolvents contains the empty clause then return true
new ←new ∪ resolvents
if new ⊆ clauses then return false
clauses ←clauses ∪new
```

Conjunctive normal form

The resolution rule applies only to clauses (that is, disjunctions of literals). A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF** .

Procedure for converting to CNF.

Example procedure for converting the sentence B1,1 ⇔ (P1,2 ∨ P2,1) into CNF.

The steps are as follows:

1. Eliminate ⇔, replacing α ⇔ β with (α ⇒ β) ∧ (β ⇒ α).

(B1,1 ⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .

2. Eliminate ⇒, replacing α ⇒ β with ¬α ∨ β:

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬(P1,2 ∨ P2,1) ∨ B1,1) .

3. CNF requires ¬ to appear only in literals, so "move ¬ inwards" by repeated application of the following equivalences :

¬(¬α) ≡ α (double-negation elimination)

¬(α ∧ β) ≡ (¬α ∨ ¬β) (De Morgan)

¬(α ∨ β) ≡ (¬α ∧ ¬β) (De Morgan)

This example, requires just one application of the last rule:

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ ((¬P1,2 ∧ ¬P2,1) ∨ B1,1) .

4. Now the sentence containing nested ∧ and ∨ operators applied to literals. Distributivity law is applied distributing ∨ over ∧ wherever possible.

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬P1,2 ∨ B1,1) ∧ (¬P2,1 ∨ B1,1) .

The original sentence is now in CNF, as a conjunction of three clauses.

Example : A grammar for conjunctive normal form, Horn clauses, and definite clauses.

CNFSentence → Clause 1 ∧· · · ∧ Clause n
Clause → Literal 1 ∨ · · ·∨ Literal m
Fact → Symbol
Literal → Symbol | ¬Symbol
Symbol → P | Q | R | . . .
Horn Clause Form → Definite Clause Form | Goal Clause Form
Definite Clause Form → Fact | (Symbol 1 ∧· · · ∧ Symbol l) ⇒ Symbol
Goal Clause Form → (Symbol 1 ∧· · · ∧ Symbol l) ⇒ False

Horn clauses and definite clauses

The completeness of resolution makes it a very important inference method. One such restricted form is the **definite clause**, which is a disjunction of literals of which *exactly one is positive*. For example, the clause (¬L1,1 ∨¬Breeze ∨B1,1) is a definite clause, whereas (¬B1,1 ∨ P1,2 ∨ P2,1) is not.

Slightly more general is the **Horn clause**, which is a disjunction of literals of which *atmost one is positive*. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called **goal clauses**. Horn clauses are closed under resolution.

Knowledge bases containing only definite clauses are interesting for three reasons:

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal.

For example, the definite clause (¬L1,1 ∨ ¬Breeze ∨ B1,1) can be written as the implication

(L1,1 ∧ Breeze) ⇒ B1,1. In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze, then [1,1] is breezy.
In Horn form, the premise is called the **body** and the conclusion is called the **head**. A sentence consisting of a single positive literal, such as L1,1, is called a **fact**.
2. Inference with Horn clauses can be done through the **forward chaining** and **backward chaining** algorithms. This type of inference is the basis for **logic programming**.
3. Deciding entailment with Horn clauses can be done in time that is *linear* in the size of the knowledge base

PROPOSITIONAL MODEL CHECKING

The two families of efficient algorithms for general propositional inference based on model checking: One approach based on backtracking search, and one on local hill-climbing search. These algorithms are part of the "technology" of propositional logic.

A complete backtracking algorithm

The first algorithm often called the **Davis Putnam algorithm**, after the seminal paper by Martin Davis and Hilary Putnam (1960). The algorithm is in fact the version described by Davis, Logemann, and Loveland (1962), hence it is known as DPLL after the initials of all four authors. DPLL takes as input a sentence in conjunctive normal form—a set of clauses. Like BACKTRACKING-SEARCH and TT-ENTAILS, it is essentially a recursive, depth-first enumeration of possible models.

It embodies three improvements over the simple scheme of TT-ENTAILS.

Early termination**:** The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if *any* literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete. For example, the sentence (A ∨ B) ∧ (A ∨ C) is true if A is true, regardless of the values of B and C. Similarly, a sentence is false if *any* clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space.

Pure symbol heuristic**:** A **pure symbol** is a symbol that always appears with the same "sign" in all clauses. For example, in the three clauses (A ∨ ¬B), (¬B ∨ ¬C), and (C ∨ A), the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure. For example, if the model contains B =false, then the clause (¬B ∨ ¬C) is already true, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.

Unit clause heuristic**:** A **unit clause** was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned false by the model. For example, if the model contains B =true, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must be set to false.

The DPLL algorithm for checking satisfiability of a sentence in propositional logic.

```
function DPLL-SATISFIABLE?(s) returns true or false
inputs: s, a sentence in propositional logic
clauses ←the set of clauses in the CNF representation of s
symbols←a list of the proposition symbols in s
return DPLL(clauses, symbols,{ })

function DPLL(clauses, symbols,model ) returns true or false
if every clause in clauses is true in model then return true
if some clause in clauses is false in model then return false
P, value←FIND-PURE-SYMBOL(symbols, clauses,model )
if P is non-null then return DPLL(clauses, symbols – P,model ∪ {P=value})
P, value←FIND-UNIT-CLAUSE(clauses,model )
if P is non-null then return DPLL(clauses, symbols – P,model ∪ {P=value})
P ←FIRST(symbols); rest ←REST(symbols)
return DPLL(clauses, rest ,model ∪ {P=true}) or
DPLL(clauses, rest ,model ∪ {P=false}))
```

Some of the tricks used for solving large problems are

1. **Component analysis** : As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called **components**, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.
2. **Variable and value ordering**: The simple implementation of DPLL uses an arbitrary variable ordering and always tries the value *true* before *false*. The **degree heuristic** suggests choosing the variable that appears most frequently over all remaining clauses.
3. **Intelligent backtracking** :Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict. All SAT solvers that do intelligent backtracking use some form of **conflict clause learning** to record conflicts so that they won't be repeated later in the search. Usually a limited-size set of conflicts is kept, and rarely used ones are dropped.
4. **Random restarts** :Sometimes a run appears not to be making progress. In this case, start over can be done from the top of the search tree, rather than trying to continue. After restarting, different random choices (in variable and value selection) are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.
5. **Clever indexing** : The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things as "the set of clauses in which variable Xi appears as a positive literal." This task is complicated by the fact that the algorithms are interested only in the clauses that have not yet been satisfied by previous assignments to variables, so the indexing structures must be updated dynamically as the computation proceeds.

Local search algorithms

These algorithms can be applied directly to satisfiability problems, provided the right evaluation function is chosen.

One of the simplest and most effective algorithms is the WALKSAT . On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip: (1) a “min-conflicts” step that minimizes the number of unsatisfied clauses in the new state and (2) a “random walk” step that picks the symbol randomly.

The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables.

```
function WALKSAT(clauses, p,max flips) returns a satisfying model or failure
inputs: clauses, a set of clauses in propositional logic
p, the probability of choosing to do a “random walk” move, typically around 0.5
max flips, number of flips allowed before giving up
model ←a random assignment of true/false to the symbols in clauses
for i= 1to max flips do
if model satisfies clauses then return model
clause←a randomly selected clause from clauses that is false in model
with probability p flip the value in model of a randomly selected symbol from clause
else flip whichever symbol in clause maximizes the number of satisfied clauses
return failure
```

WALKSAT is most useful when a solution is expected to exist. On the other hand, WALKSAT cannot always detect *unsatisfiability,* which is required for deciding entailment.For example, an agent cannot *reliably* use WALKSAT to prove that a square is safe in the wumpus world. Instead, it can say, “I thought about it for an hour and couldn’t come up with a possible world in which the square *isn’t* safe.” This may be a good empirical indicator that the square is safe, but it’s certainly not a proof.

AGENTS BASED ON PROPOSITIONAL LOGIC

The first step is to enable the agent to deduce, to the extent possible, the state of the world given its percept history. This requires writing down a complete logical model of the effects of actions. Next is how the agent can keep track of the world efficiently without going back into the percept history for each inference. Finally, how the agent can use logical inference to construct plans that are guaranteed to achieve its goals

The current state of the world

A logical agent operates by deducing what to do from a knowledge base of sentences about the world. The knowledge base is composed of axioms—general knowledge about how the world works—and percept sentences obtained from the agent’s experience in a particular world

For example,The agent knows that the starting square contains no pit ($\neg P1,1$) and no wumpus ($\neg W1,1$). Furthermore, for each square, it knows that the square is breezy if and only if a neighboring square has a pit; and a square is smelly if and only if a neighboring square has a wumpus. Thus, a large collection of sentences can be of the following form:

$B1,1 \Leftrightarrow (P1,2 \lor P2,1)$
$S1,1 \Leftrightarrow (W1,2 \lor W2,1)$
$\cdot \quad \cdot \quad \cdot$

The agent also knows that there is exactly one wumpus. This is expressed in two parts. First, there is *at least one* wumpus:

$W1,1 \lor W1,2 \lor \cdot \cdot \cdot \lor W4,3 \lor W4,4$.

Then, there is *at most one* wumpus. For each pair of locations, add a sentence saying that at least one of them must be wumpus-free:

$\neg W1,1 \lor \neg W1,2$

$\neg W1,1 \lor \neg W1,3$

$\cdot \quad \cdot \quad \cdot$

$\neg W4,3 \lor \neg W4,4$.

Now let's consider the agent's percepts. If there is currently a stench, one might suppose that a proposition Stench should be added to the knowledge base. This is not quite right, however: if there was no stench at the previous time step, then $\neg$Stench would already be asserted, and the new assertion would simply result in a contradiction.

The idea of associating propositions with time steps extends to any aspect of the world that changes over time. the word **fluent** to refer an aspect of the world that changes. "Fluent" is a synonym for "state variable. Symbols associated with permanent aspects of the world do not need a time superscript and are sometimes called **atemporal variables**.

To describe how the world changes,   **effect axioms are written** that specify the outcome of an action at the next time step.
For example, if the agent is at location [1, 1] facing east at time 0 and goes Forward , the result is that the agent is in square [2, 1] and no longeris in [1, 1]:
$L^0 1,1 \land FacingEast^0 \land Forward^0 \Rightarrow (L^1 2,1 \land \neg L^1 1,1)$ ………………….(1)

Suppose that the agent does decide to move *Forward* at time 0 and asserts this fact into its knowledge base. Given the effect axiom in Equation(1), combined with the initial assertions about the state at time 0, the agent can now deduce that it is in [2, 1]. That is, ASK(KB, $L^1 2,1$) =true. So far, so good.if ASK(KB, HaveArrow1), the answer is false, that is, the agent cannot prove it still has the arrow; nor can it prove it *doesn't* have it! The information has been lost because the effect axiom fails to state what remains *unchanged* as the result of an action. The need to do this gives rise to the **frame problem**. One possible solution to the frame problem would be to add **frame axioms** explicitly asserting all the propositions that remain the same.

In a world with m different actions and n fluents, the set of frame axioms will be of size O(mn). This specific manifestation of the frame problem is sometimes called the **representational frame problem**.

Solving the representational frame problem requires defining the transition model with a set of axioms of size O(mk) rather than size O(mn). There is also an **inferential frame problem**: the problem of projecting forward the results of a t step plan of action in time O(kt) rather than O(nt).

Specifying all the exceptions is called the **qualification problem**. There is no complete solution within logic; system designers have to use good judgment in deciding how detailed they want to be in specifying their model, and what details they want to leave out.

A hybrid agent

The ability to deduce various aspects of the state of the world can be combined fairly straightforwardly with condition–action rules and with problem-solving algorithms to produce a **hybrid agent** for the wumpus world.

A hybrid agent program for the wumpusworld.

```
function HYBRID-WUMPUS-AGENT(percept ) returns an action
inputs: percept , a list, [stench,breeze,glitter ,bump,scream]
persistent: KB, a knowledge base, initially the atemporal “wumpus physics”
t , a counter, initially 0, indicating time
plan, an action sequence, initially empty
TELL(KB,MAKE-PERCEPT-SENTENCE(percept , t ))
TELL the KB the temporal “physics” sentences for time t
safe ←{[x , y] : ASK(KB,OKt
x,y) = true}
if ASK(KB,Glitter t) = true then
plan ←[Grab] + PLAN-ROUTE(current , {[1,1]}, safe) + [Climb]
if plan is empty then
unvisited ←{[x , y] : ASK(KB,Lt_
x,y) = false for all t_ ≤ t}
plan ←PLAN-ROUTE(current, unvisited ∩safe, safe)
if plan is empty and ASK(KB,HaveArrowt) = true then
possible wumpus ←{[x , y] : ASK(KB,¬Wx,y) = false}
plan ←PLAN-SHOT(current , possible wumpus, safe)
if plan is empty then // no choice but to take a risk
not unsafe ←{[x , y] : ASK(KB,¬ OKtx,y) = false}
plan ←PLAN-ROUTE(current, unvisited ∩not unsafe, safe)
if plan is empty then
plan ←PLAN-ROUTE(current, {[1, 1]}, safe) + [Climb]
action ←POP(plan)
TELL(KB,MAKE-ACTION-SENTENCE(action, t ))
t ←t + 1
return action
```

```
function PLAN-ROUTE(current,goals,allowed) returns an action sequence
inputs: current , the agent's current position
goals, a set of squares; try to plan a route to one of them
allowed, a set of squares that can form part of the route
problem ←ROUTE-PROBLEM(current , goals,allowed)
return A*-GRAPH-SEARCH(problem)
```

The agent program maintains and updates a knowledge base as well as a current plan. The initial knowledge base contains the *atemporal* axioms—those that don't depend on t, such as the axiom relating the breeziness of squares to the presence of pits. At each time step, the new percept sentence is added along with all the axioms that depend on t, such as the successor-state axioms. Then, the agent uses logical inference, by ASKing questions of the knowledge base, to work out which squares are safe and which have yet to be visited.

The main body of the agent program constructs a plan based on a decreasing priority of goals. First, if there is a glitter, the program constructs a plan to grab the gold, follow a route back to the initial location, and climb out of the cave. Otherwise, if there is no current plan, the program plans a route to the closest safe square that it has not visited yet, making sure the route goes through only safe squares. Route planning is done with A∗ search, not with ASK. If there are no safe squares to explore, the next step—if the agent still has an arrow—is to try to make a safe square by shooting at one of the possible wumpus locations. These are determined by asking where ASK(KB, ¬ Wx,y) is false—that is, where it is *not* known that there is *not* a wumpus. The function PLAN-SHOT (not shown) uses PLAN-ROUTE to plan a sequence of actions that will line up this shot. If this fails, the program looks for a square to explore that is not provably unsafe—that is, a square for which ASK(KB, ¬ OKtx,y) returns false. If there is no such square, then the mission is impossible and the agent retreats to [1, 1] and climbs out of the cave.

Logical state estimation

The agent program has one major weakness: as time goes by, the computational expense involved in the calls to ASK goes up and up. This happens mainly because the required inferences have to go back further and further in time and involve more and more proposition symbols. The obvious answer is to save, or **cache**, the results of inference, so that the inference process at the next time step can build on the results of earlier steps instead of having to start again from scratch.

The past history of percepts and all their ramifications can be replaced by the **belief state**—that is, some representation of the set of all possible current states of the world.The process of updating the belief state as new percepts arrive is called **state estimation**. A belief state is normally an explicit list of states, here it is a logical sentence involving the proposition symbols associated with the current time step, as well as the atemporal symbols.
For example, the logical sentence
WumpusAlive1 ∧ L12,1 ∧ B2,1 ∧ (P3,1 ∨ P2,2)
represents the set of all states at time 1 in which the wumpus is alive, the agent is at [2, 1], that square is breezy, and there is a pit in [3, 1] or [2, 2] or both.

One very common and natural scheme for *approximate* state estimation is to represent belief states as conjunctions of literals, that is, 1-CNF formulas. To do this, the agent program simply tries to prove Xt and ¬ Xt for each symbol Xt (as well as each atemporal symbol whose truth value is not yet known), given the belief state at $t - 1$. The conjunction of provable literals becomes the new belief state, and the previous belief state is discarded. Thus, *the set of possible states represented by the 1-CNF belief state includes all states that are in fact possible given the full percept history.*

Making plans by propositional inference

The agent inuses logical inference to determine which squares are safe, but uses A∗ search to make plans.

The basic idea is very simple:

1. Construct a sentence that includes

(a) $Init^0$, a collection of assertions about the initial state;

(b) $Transition^1, \ldots, Transition^t$, the successor-state axioms for all possible actions at each time up to some maximum time t;

(c) the assertion that the goal is achieved at time t: $HaveGold^t \wedge ClimbedOut^t$.

2. Present the whole sentence to a SAT solver. If the solver finds a satisfying model, then the goal is achievable; if the sentence is unsatisfiable, then the planning problem is impossible.

3. Assuming a model is found, extract from the model those variables that represent actions and are assigned true. Together they represent a plan to achieve the goals.

A propositional planning procedure, SATPLAN, is shown in Figure .

```
function SATPLAN(init , transition, goal , T max) returns solution or failure
inputs: init , transition, goal , constitute a description of the problem
T max, an upper limit for plan length
for t = 0 to T max do
cnf ←TRANSLATE-TO-SAT(init , transition, goal , t )
model ←SAT-SOLVER(cnf )
if model is not null then
return EXTRACT-SOLUTION(model )
return failure
```

It implements the basic idea just given, with one twist. Because the agent does not know how many steps it will take to reach the goal, the algorithm tries each possible number of steps t, up to some maximum conceivable plan length Tmax. In this way, it is guaranteed to find the shortest plan if one exists. Because of the way SATPLAN searches for a solution, this approach cannot be used in a partially observable environment; SATPLAN would just set the unobservable variables to the values it needs to create a solution.

To summarize, SATPLAN finds models for a sentence containing the initial state, the goal, the successor-state axioms, the precondition axioms, and the action exclusion axioms. It can be shown that this collection of axioms is sufficient, in the sense that there are no longer any spurious “solutions.” Any model satisfying the propositional sentence will be a valid plan for the original problem.

FIRST ORDER PREDICATE LOGIC/FIRST ORDER LOGIC/FOL

Predicate logic is an extension of propositional logic .It adds the concepts of predicates and quantifiers to better capture the meaning of statements that cannot be adequately expressed by propositional logic

FOL is a symbolized reasoning in which each sentence or statement is broken down into subject and a predicate.

In FOL there are two important elements

- Objects
- Relations

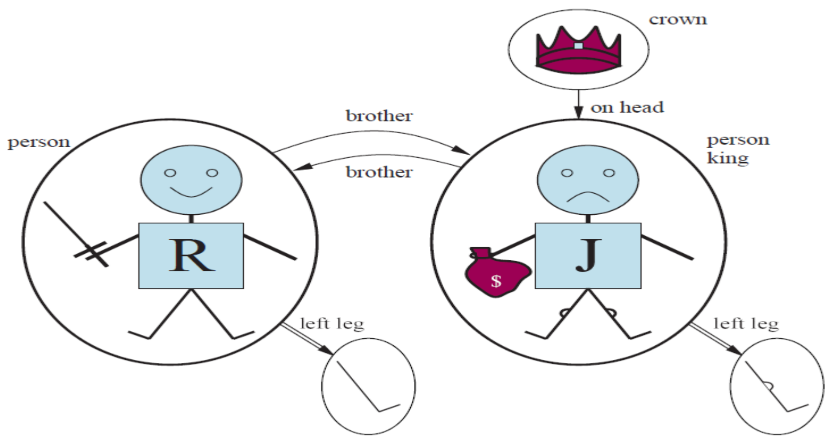
Noun and noun phrases makes object eg: house, people, colour. Verb and verb phrases makes relation These can be unary relations or properties such as red, round or n-ary relations such as brother of ,bigger than…Relation takes objects as argument and generates truth values. Eg: x is father of y. Function is a type of relation which takes object as argument and generates another object as value eg: father of y

SYNTAX AND SEMANTICS OF FIRST ORDER PREDICATE LOGIC

Models for first-order logic

Models for first-order logic have objects in them. The **domain** of a model is the set of objects or **domain elements** it contains. The domain is required to be *nonempty*—every possible world must contain at least one object. Figure shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown. The objects in the model may be *related* in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation TUPLE is just the set of **tuples** of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set
{ <Richard the Lionheart, King John>, <King John, Richard the Lionheart> } .

Figure : A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).



The model also contains unary relations, or properties: the "person" property is true of both Richard and John; the "king" property is true only of John and the "crown" property is true only of the crown.
Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:
<Richard the Lionheart>→ Richard's left leg
<King John> → John's left leg.

Strictly speaking, models in first-order logic require **total functions**, that is, there must be a value for every input tuple. Thus, the crown must have a left leg and so must each of the left legs.

Symbols and interpretations

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: **constant symbols**, which stand for objects; **predicate symbols**, which stand for relations; and **function symbols**, which stand for functions

Terms

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object.

Atomic sentences

Objects and predicate symbols for referring to relations, can be together to make **atomic sentences** that state facts. An **atomic sentence** (or **atom** for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as
Brother (Richard , John).
*An atomic sentence is **true** in a given model if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.*

Complex sentences

Logical connectives can be used to construct more complex sentences, with the same syntax and semantics as in propositional calculus.
Eg:
¬ Brother (LeftLeg(Richard), John)
Brother (Richard , John) ∧ Brother (John,Richard)
King(Richard) ∨ King(John)
¬ King(Richard) ⇒ King(John)

Figure: The syntax of first-order logic with equality, specified in Backus–Naur form
Sentence → AtomicSentence | ComplexSentence
AtomicSentence → Predicate | Predicate(Term, . . .) | Term = Term
ComplexSentence → (Sentence)
| ¬ Sentence
| Sentence ∧ Sentence

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

Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

Quantifiers :
Quantifiers is used to express the extent to which a predicate is true over a range of elements. First-order logic contains two standard quantifiers, called *universal* and *existential*.

Universal quantification (∀)
Mathematical statements sometimes says that a property is true for all values of a variable in a particular domain ,such a statement is expressed using Universal Quantifier
Eg:"All kings are persons," is written in first-order logic as
∀ x King(x) ⇒ Person(x) .
∀ is usually pronounced "For all . . .".
Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called **variable**. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function—for example, LeftLeg(x). A term with no variables is called a **ground term**

Existential quantification (∃)
Mathematical statements sometimes says that a property is true for some values of a variable in a particular domain ,such a statement is expressed using **Existential** Quantifier
Eg:Some men are honest

∃ x Man(x) ⇒ Honest(x) .

Nested quantifiers
More complex sentences can be expressed using multiple quantifiers.
i)The simplest case is where the quantifiers are of the same type.
For example, "Brothers are siblings" can be written as
∀ x ∀ y Brother (x, y) ⇒ Sibling(x, y) .
ii)Consecutive quantifiers of the same type can be written as one quantifier with several variables.
For example, to say that siblinghood is a symmetric relationship,
∀ x, y Sibling(x, y) ⇔ Sibling(y, x) .
iii)In other cases quantifiers may be of different type.
"Everybody loves somebody" means that for every person, there is someone that person loves:

∀ x ∃ y Loves(x, y) .
On the other hand, to say "There is someone who is loved by everyone,"
∃ y ∀ x Loves(x, y) .
The order of quantification is therefore very important

Connections between ∀ and ∃
The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes bitterguard is the same as asserting there does not exist someone who likes them, and vice versa:

∀ x ¬ Likes(x, bitterguard) is equivalent to ¬ ∃ x Likes(x, bitterguard) .

"Everyone likes ice cream" means that there is no one who does not like ice cream:

∀ x Likes(x, IceCream) is equivalent to ¬ ∃ x ¬ Likes(x, IceCream) .

Because ∀ is really a conjunction over the universe of objects and ∃ is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$\forall x\ \neg P \equiv \neg \exists x\ P \quad \neg(P \lor Q) \equiv \neg P \land \neg Q$

$\neg \forall x\ P \equiv \exists x\ \neg P \quad \neg(P \land Q) \equiv \neg P \lor \neg Q$

$\forall x\ P \equiv \neg \exists x\ \neg P \quad P \land Q \equiv \neg(\neg P \lor \neg Q)$

$\exists x\ P \equiv \neg \forall x\ \neg P \quad P \lor Q \equiv \neg(\neg P \land \neg Q)$.

Equality
First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. The **equality symbol is used** to signify that two terms refer to the same object. For example,
Father (John)=Henry
says that the object referred to by Father (John) and the object referred to by Henry are the same. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, can be written as

∃ x, y Brother (x,Richard) ∧ Brother (y,Richard) ∧ ¬ (x=y) .

The sentence ∃ x, y Brother (x,Richard) ∧ Brother (y,Richard)
does not have the intended meaning. To see this, consider the extended interpretation in which both x and y are assigned to King John. The addition of ¬ (x=y) rules out such models. The notation x ≠y is sometimes used as an abbreviation for ¬ (x=y).

KNOWLEDGE REPRESENTATION AND ENGINEERING
The general process of knowledge-base construction —a process called **knowledge engineering**. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain.

The knowledge engineering process
Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. *Identify the task(questions).* The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance.
For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents

2. *Assemble the relevant knowledge.* The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called **knowledge acquisition**. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

3. *Decide on a vocabulary of predicates, functions, and constants.* That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering *style*. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain. The word *ontology* means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine
their specific properties and interrelationships.

4. *Encode general knowledge about the domain.* The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

5. *Encode a description of the specific problem instance.* If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.

6. *Pose queries to the inference procedure and get answers.* This is where the reward is: the inference procedure operate on the axioms and problem-specific facts to derive the facts. Thus, the need for writing an application-specific solution algorithm can be avoided.

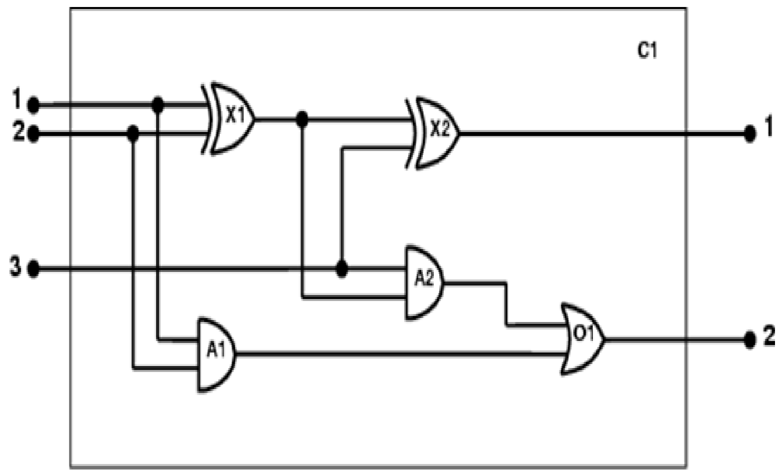
7. *Debug the knowledge base.* The answers to queries will seldom be correct on the first try. More precisely, the answers will be correct *for the knowledge base as written*, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting.

For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue. Missing axioms or axioms that are too weak can be easily identified by noticing places where the chain of reasoning stops unexpectedly.

For example, if the knowledge base includes a diagnostic rule for finding the wumpus,
∀ s Smelly(s) ⇒ Adjacent (Home(Wumpus), s) ,
instead of the biconditional, then the agent will never be able to prove the *absence* of wumpuses. Incorrect axioms can be identified because they are false statements about the world.

The electronic circuits domain

The seven step process is explained below for electronic circuits domain



Identify the task: -
o Analyse the circuit functionality, does the circuit actually add properly? (Circuit Verification).
Assemble the relevant knowledge: -
o The circuit is composed of wire and gates.
o The four types of gates (AND, OR, NOT, XOR) with two input terminals and one output terminal knowledge is collected.

Decide on a vocabulary: -
- The functions, predicates and constants of the domain are identified.
- Functions are used to refer the type of gate. Type (x1) = XOR,

where x1 ---- Name of the gate and Type ---------------- function
- The same can be represented by either binary predicate (or) individual type predicate.

Type (x1, XOR) – binary predicate
XOR (x1) – Individual type
- A gate or circuit can have one or more terminals. For x1, the terminals are x1In1, x1In2 and x1 out1

Where x1 In1 -----1st input of gate x1
x1 In2 -------2nd input of gate x1
x1 out1 -------------------- output of gate x1

- Then the connectivity between the gates represented by the predicate connected. (i.e.) connected (out(1, x1), In(1,x2)).
- Finally the possible values of the output terminal C1, as true or false, represented as a signal with 1 or 0.

Encode general knowledge of the domain:-

o This example needs only seven simple rules to describe everything need to know about circuits
1. If two terminals are connected, then they have the same signal:
$\forall$ t1, t2 Terminal (t1) $\wedge$ Terminal (t2) $\wedge$ Connected(t1, t2) $\Rightarrow$ Signal (t1)=Signal (t2) .
2. The signal at every terminal is either 1 or 0:
$\forall$ t Terminal (t) $\Rightarrow$ Signal (t)=1 $\vee$ Signal (t)=0 .
3. Connected is commutative:
$\forall$ t1, t2 Connected(t1, t2) $\Leftrightarrow$ Connected(t2, t1) .
4. There are four types of gates:
$\forall$ g Gate(g) $\wedge$ k = Type(g) $\Rightarrow$ k = AND $\vee$ k = OR $\vee$ k = XOR $\vee$ k = NOT .
5. An AND gate's output is 0 if and only if any of its inputs is 0:
$\forall$ g Gate(g) $\wedge$ Type(g)=AND $\Rightarrow$Signal (Out(1, g))= 0 $\Leftrightarrow$ $\exists$n Signal (In(n, g))=0 .
6. An OR gate's output is 1 if and only if any of its inputs is 1: $\forall$ g Gate(g) $\wedge$ Type(g)=OR $\Rightarrow$ Signal (Out(1, g))= 1 $\Leftrightarrow$ $\exists$n Signal (In(n, g))=1 .
7. An XOR gate's output is 1 if and only if its inputs are different:
$\forall$ g Gate(g) $\wedge$ Type(g)=XOR $\Rightarrow$Signal (Out(1, g))= 1 $\Leftrightarrow$ Signal (In(1, g)) $\neq$ Signal (In(2, g)) .
8. A NOT gate's output is different from its input:
$\forall$ g Gate(g) $\wedge$ (Type(g)=NOT) $\Rightarrow$Signal (Out(1, g)) $\neq$ Signal (In(1, g)) .

Encode the specific problem instance:

o First, the gates are categorized:
Type(X1) = XOR
Type(X2) = XOR
Type(A1) = AND
Type(A2) = AND
Type(O1) = OR
Then, the connections between them are shown
Connected(Out(1,X1),In(1,X2))
Connected(In(1,C1),In(1,X1))
Connected(Out(1,X1),In(2,A2))
Connected(In(1,C1),In(1,A1))
Connected(Out(1,A2),In(1,O1))
Connected(In(2,C1),In(2,X1))
Connected(Out(1,A1),In(2,O1))
Connected(In(2,C1),In(2,A1))

Pose Queries to the inference procedure:

o What combinations of inputs would cause the first output of C1(the sum bit) to be 0 and the second output of C1 (the carr y bit) to be 1?

o ∃ i1, i2, i3 Signal (In(1, C1))=i1 ∧ Signal (In(2, C1))=i2 ∧ Signal (In(3, C1))=i3∧ Signal (Out(1, C1))=0 ∧ Signal (Out(2, C1))=1 .

o The answers are substitutions for the variables i1,i2 and i3 Such that the resulting sentence is entailed by the knowledge base.

o There are three such substitutions as: { i1/1,i2/1 , i3/0} { i1/1,i2/0 , i3/1} { i1/0,i2/1 ,i3/1}

Debug the knowledge base:

o The knowledge base is checked with different constraints.

o For Example:- if the assertion 1 ≠ 0 is not included in the knowledge base then it is variable to prove any output for the circuit, except for the input cases 000 and 110.

INFERENCES IN FIRST-ORDER LOGIC

Simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers. These rules lead naturally to the idea that *first-order* inference can be done by converting the knowledge base to *propositional* logic and using *propositional* inference

Inference rules for quantifiers

Suppose the knowledge base contains the standard axiom stating that all greedy kings are evil:
∀ x King(x) ∧ Greedy(x) ⇒ Evil(x) .
Then it seems quite permissible to infer any of the following sentences:
King(John) ∧ Greedy(John) ⇒ Evil(John)
King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)
King(Father (John)) ∧ Greedy(Father (John)) ⇒ Evil(Father (John)) .

The rule of **Universal Instantiation**(**UI** for short) says that any sentence obtained can be inferred by substituting a **ground term** (a term without variables) for the variable.
To write out the inference rule formally, the notion of **substitutions is** introduced

Let SUBST(θ,α) denote the result of applying the substitution θ to the sentence α. Then the rule is written

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

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions {x/John}, {x/Richard }, and {x/Father (John)}.
In the rule for **Existential Instantiation**, the variable is replaced by a single *new constant symbol*. The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \ \ \alpha}{\textsc{Subst}(\{v/k\}, \alpha)} \ .$$

For example, from the sentence
∃ x Crown(x) ∧ OnHead(x, John)
The following sentence cab be inferred
Crown(C1) ∧ OnHead(C1, John)

Reduction to propositional inference

Once there are rules for inferring non-quantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of *all possible* instantiations.

For example, suppose the knowledge base contains just the sentences
∀ x King(x) ∧ Greedy(x) ⇒ Evil(x)
King(John)
Greedy(John)
Brother (Richard, John) .

Then UI is applied to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case, {x/John} and {x/Richard }. Hence

King(John) ∧ Greedy(John) ⇒ Evil(John)
King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard) , is obtained

and then universally quantified sentence is discarded. Now, the knowledge base is essentially propositional if the ground atomic sentences is viewed—King(John), Greedy(John), and so on—as proposition symbols. Therefore, any of the complete propositional algorithms can be applied to obtain conclusions such as Evil(John).This technique of **propositionalization** can be made completely general.

UNIFICATION

- 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.
- Unification is a "pattern matching" procedure that takes two atomic sentences, called literals, as input, and returns "failure" if they do not match and a substitution list, Theta, if they do match.
- Theta is called the most general unifier (mgu)
- The unification routine, UNIFY is to take two atomic sentences p and q and returns α substitution that would make p and q look the same
- UNIFY (p, q) = θ where SUBST (θ, p) = SUBST (θ, q) θ = Unifier of two sentences
- For example:-

p – S1(x, x)

q– S1(y, z)

Assume θ = y

p – S1(y, y) – x/y (Substituting y for x)

q – S1(y, y) – z/y (Substituting y for z)

- In the above two sentences (p, q), the unifier of two sentences (i.e.) θ = y is substituted in both the sentences, which derives a same predicate name, same number of arguments and same argument names.
- Therefore the given two sentences are unified sentences.
- The function UNIFY will return its result as fail, for two different types of criteria's as follows,
  ✓ If the given two atomic sentences (p, q) are differs in its predicate name then the UNIFY will return failure as a result

For Example: - **p – hate (M, C), q – try (M, C)**

✓ If the given two atomic sentences (p, q) are differs in its number of arguments then the UNIFY will return failure as a result

For Example: - **p – try (M, C), q – try (M, C, R)**

- For Example: - The Query is **Knows (John, x) whom does John Know?**
- Some answers to the above query can be found by finding all sentences in the KB that unify with knows (John, x)
- Assume the KB has as follows,
- Knows (John, John) Knows (y, Leo) Knows (y, Mother(y)) Knows (x, Elizabeth)
- The results of unification are as follows,

- UNIFY (knows (john, x), knows (John, Jane)) = {x/Jane}
- UNIFY (knows (john, x), knows (y, Leo)) = {x/Leo, y/John}
- UNIFY (knows (john, x), knows (y, Mother(y)) = {y/John, x/Mother (John)} UNIFY (knows (john, x), knows (x, Elizabeth)) = fail
- x cannot take both the values John and Elizabeth at the same time.
- Knows (x, Elizabeth) means "Everyone knows Elizabeth" from this we able to infer that John knows Elizabeth.
- This can be avoided by using standardizing apart one of the two sentences being unified (i.e.) renaming is done to avoid name clashes.
- For Example:-
- UNIFY (Knows (john, x), knows (x_1, Elizabeth)) = {x/Elizabeth, x_1/John}

Algorithm:

```
Algorithm: Unify(L1, L2)
Step. 1: If L1 or L2 is a variable or constant, then:
        a) If L1 or L2 are identical, then return NIL.
        b) Else if L1  is a variable,
                a. then if L1 occurs in L2, then return FAILURE
                b. Else return { (L2/ L1)}.
        c) Else if L2 is a variable,
                a. If L2 occurs in L1 then return FAILURE,
                b. Else return {( L1/ L2)}.
        d) Else return FAILURE.
Step.2: If the initial Predicate symbol in L1 and L2 are not same, then return FAILURE.
Step. 3: IF L1 and L2 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 L1.
        a) Call Unify function with the ith element of L1 and ith element of L2, 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.
```

Example :

Marcus was a a man
man(Marcus)
2. Marcus was a Pompeian
Pompeian(Marcus)
3. All Pompeians were Romans
∀x (Pompeians(x) → Roman(x))
4. Caesar was ruler
Ruler(Caesar)
5. All Romans were either loyal to Caesar or hated him.
∀x (Roman(x) → loyalto(x,Caesar) v hate(x,Caesar))
6. Everyone is loyal to someone
∀x ∃y (person(x) → person(y) ^ loyalto(x,y))
7. People only try to assassinate rulers they are not loyal to
∀x ∀y (person(x) ^ ruler(y)^ tryassassinate(x,y) → ¬ loyalto(x,y))
8. Marcus tried to assassinate Caesar
tryassassinate(Marcus, Caesar)
9. All men are persons
∀x (max(x) → person(x))

Trace the operation of the unification algorithm on each of the following pairs of literals:
i) f(Marcus) and f(Caesar)
ii) f(x) and f(g(y))
iii) f(marcus,g(x,y)) and f(x,g(Caesar, Marcus))

In propositional logic it is easy to determine that two literals can not both be true at the same time. Simply look for L and ~L. In predicate logic, this matching process is more complicated, since bindings of variables must be considered.

For example man (john) and man(john) is a contradiction while man (john) and man(Himalayas) is not. Thus in order to determine contradictions we need a matching procedure that compares two literals and discovers whether there exist a set of substitutions that makes them identical. There is a recursive procedure that does this matching. It is called Unification algorithm.

In Unification algorithm each literal is represented as a list, where first element is the name of a predicate and the remaining elements are arguments. The argument may be a single element (atom) or may be another list. For example we can have literals as
(tryassassinate Marcus Caesar)
(tryassassinate Marcus (ruler of Rome))

To unify two literals, first check if their first elements re same. If so proceed. Otherwise they can not be unified. For example the literals
(try assassinate Marcus Caesar)
(hate Marcus Caesar)
Can not be Unfied. The unification algorithm recursively matches pairs of elements, one pair at a time. The matching rules are :
i) Different constants, functions or predicates can not match, whereas identical ones can.
ii) A variable can match another variable, any constant or a function or predicate expression, subject to the condition that the function or [predicate expression must not contain any instance of the variable being matched (otherwise it will lead to infinite recursion).
iii) The substitution must be consistent. Substituting y for x now and then z for x later is inconsistent (a substitution y for x written as y/x).
The Unification algorithm is listed below as a procedure UNIFY (L1, L2). It returns a list representing the composition of the substitutions that were performed during the match. An empty list NIL indicates that a match was found without any substitutions. If the list contains a single value F, it indicates that the unification procedure failed.

FORWARD CHAINING-BACKWARD CHAINING

Forward Chaining Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.

The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

Properties of Forward-Chaining:

o It is a down-up approach, as it moves from bottom to top.

o It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.

o Forward-chaining approach is also called as data-driven as we reach to the goal using available data.

o Forward -chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Consider the following famous example which can be used in both approaches:

- **Situation: -** 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.
- **Solution: -** The given description is splitted into sentences and is converted into its corresponding FOL representation.

✓ It is a crime for an American to sell weapons to hostile nations:

$\forall x\, y\, z\; American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z \Rightarrow Criminal(x)$

✓ Nono … has some missiles,

$\exists x\; Owns(Nono,x) \wedge Missile(x)$

- ✓ all of its missiles were sold to it by Colonel West

∀x Missiles(x) ∧Owns(Nono,x) ⇒Sells(West,x,Nono)

- ✓ missiles are also weapons

∀x Missile(x) ⇒Weapon(x)

- ✓ An enemy of America counts as "hostile“

∀x Enemy(x,America) ⇒Hostile(x)

- ✓ West, who is American …

American(West)

- ✓ Nono, is a nation

Nation (Nono)

- ✓ Nono, an enemy of America

Enemy (Nono, America)

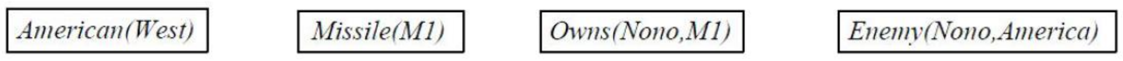
- ✓ America is nation

Nation (America)

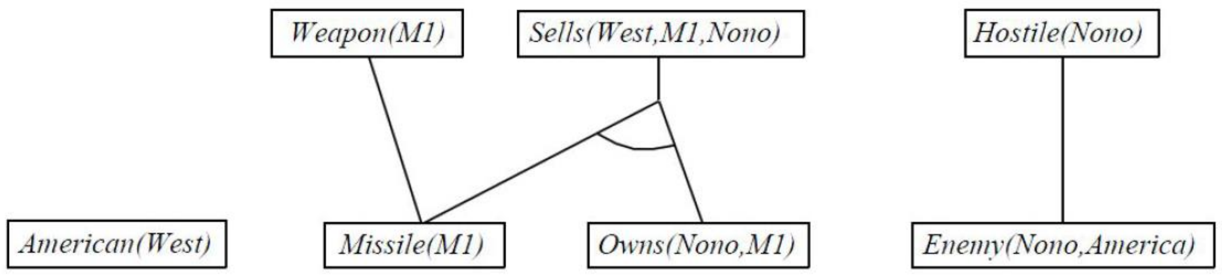
The implication sentences are (i), (iv), (v), (vi)

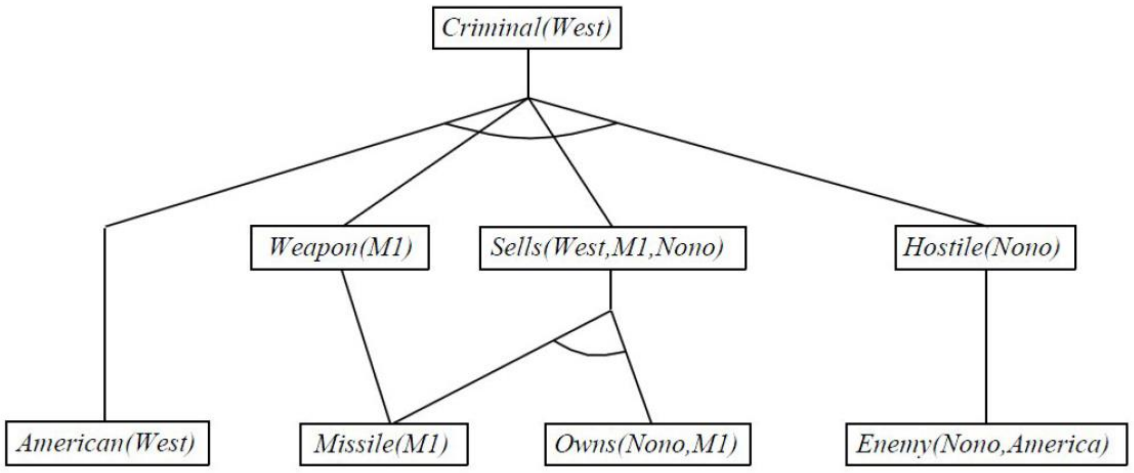
- ✓ Two iterations are required:
- ✓ On the first iteration,
  - o Step (i) has unsatisfied premises
  - o Step (iv) is satisfied with {x/M1} and sells (west, M1,Nono) is added
  - o Step (v) is satisfied with {x/m1} and weapon (M1) is added
  - o Step (vi) is satisfied with {x/Nono}, and Hostile (Nono) is added
- ✓ On the second iteration,
  - o Step (i) is satisfied with {x/West, y/M1, z/Nono} and Criminal(west) is added.

In the first step, start from the known facts and will choose the sentences which do not have implications. All these facts will be represented as below.



At the second step, the facts which infer from available facts with satisfied premises are represented.





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

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
return FOL-BC-OR(KB, query, { })
function FOL-BC-OR(KB, goal , θ) returns a substitution
for each rule in FETCH-RULES-FOR-GOAL(KB, goal ) do
(lhs ⇒ rhs)←STANDARDIZE-VARIABLES(rule)
for each θ′ in FOL-BC-AND(KB, lhs,UNIFY(rhs, goal , θ)) do
yield θ′
function FOL-BC-AND(KB, goals, θ) returns a substitution
if θ = failure then return
else if LENGTH(goals) = 0 then yield θ
else
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′′
```

Backward Chaining:-

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

Properties of backward chaining:

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.

- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a **depth-first search** strategy for proof.

Example:

In backward-chaining, the same above example can be used by rewriting all the rules.

✓ It is a crime for an American to sell weapons to hostile nations:

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

✓ Nono … has some missiles,

∃x Owns(Nono,x) ∧Missile(x)

✓ all of its missiles were sold to it by Colonel West

∀x Missiles(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)

✓ Missiles are weapons

∀x Missile(x) ⇒Weapon(x)

✓ An enemy of America counts as "hostile“

∀x Enemy(x,America) ⇒Hostile(x)

✓ West, who is American …

American(West)

✓ Nono, is a nation

Nation (Nono)

✓ Nono, an enemy of America

Enemy (Nono, America)

✓ America is nation

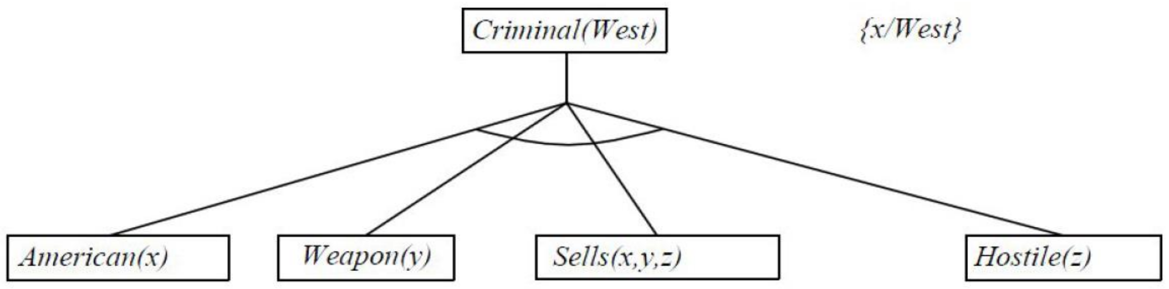
Nation (America)

Step-1: At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Colonel west is Criminal," so following is the predicate of it.

$\boxed{Criminal(West)}$

Step-2:

At the second step, we will infer other facts form goal fact which satisfies the rules. So as we can see in Rule-1, the goal predicate Criminal (west) is present with substitution {x/west, y/m1, z/nano}. So we will add all the conjunctive facts below the first level and will replace x with west.



Step-3:

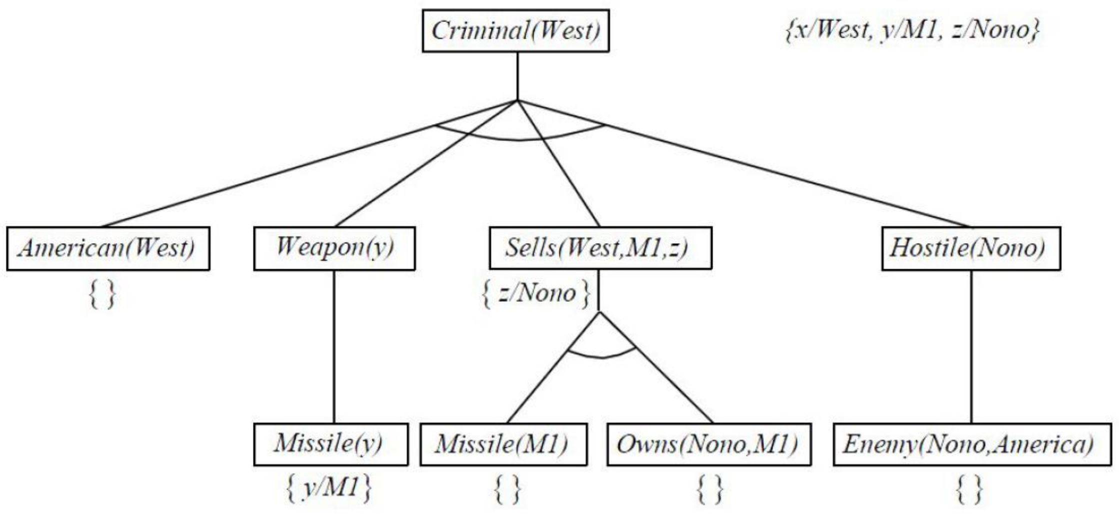
At step-3, we will extract further fact Missile(M1) which infer from Weapon(y), as it satisfies Rule-(4).

Step-4:

At step-4, we can infer facts Missile(m1) and Owns(nano, m1) form Sells(west, m1, z) which satisfies the **Rule- 3**, with the substitution of nano in place of z. So these two statements are proved here.

Step-5:

At step-5, we can infer the fact **Enemy(nano, America)** from **Hostile(nano)** which satisfies Rule- 5. And hence all the statements are proved true using backward chaining.



Backward chaining is designed to find all answers to a question asked to the knowledge base. Therefore it requires a **ASK** procedure to derive the answer.

The procedure **BACK WARD-CHAIN** will check two constraints.

✓ If the given question can derive a answer directly from the sentences of the knowledge base then it returns with answers.

✓ If the first constraint is not satisfied then it finds all implications whose conclusion unifies with the query and tries to establish the premises of those implications. If the premise is a conjunction then BACK-CHAIN processes the conjunction conjunct by conjunct, building up the unifiers for the whole premises as it goes

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

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
return FOL-BC-OR(KB, query, { })
function FOL-BC-OR(KB, goal , θ) returns a substitution
for each rule in FETCH-RULES-FOR-GOAL(KB, goal ) do
```

```
(lhs ⇒ rhs)←STANDARDIZE-VARIABLES(rule)
for each θ′ in FOL-BC-AND(KB, lhs,UNIFY(rhs, goal , θ)) do
yield θ′
function FOL-BC-AND(KB, goals, θ) returns a substitution
if θ = failure then return
else if LENGTH(goals) = 0 then yield θ
else
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′′
```

RESOLUTION

- Resolution is a procedure which operates on statements and finds its correctness .Resolution produces proofs by refutation .Refutation is a action of proving a statements to be false .
- A resolution refutation proof is "proof by contradiction" using resolution
- Resolution requires sentences to be in conjunctive normal form(CNF) i.e. a conjunction of clauses, where each clause is a disjunction of literals.

Consider the example:

"Everyone who loves all animals is loved by someone," or

$\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y\ Loves(y, x)]$.

The steps are as follows:

• **Eliminate implications**:

$\forall x\ [\neg \forall y\ \neg Animal(y) \lor Loves(x, y)] \lor [\exists y\ Loves(y, x)]$.

• **Move ¬ inwards**: In addition to the usual rules for negated connectives, the rules for negated quantifiers are

$\neg \forall x\ p$ becomes $\exists\ x\ \neg p$

$\neg \exists x\ p$ becomes $\forall\ x\ \neg p$.

Thus the sentence goes through the following transformations:

$\forall x\ [\exists y\ \neg(\neg Animal(y) \lor Loves(x, y))] \lor [\exists y\ Loves(y, x)]$.

$\forall x\ [\exists y\ \neg \neg Animal(y) \land \neg Loves(x, y)] \lor [\exists y\ Loves(y, x)]$.

$\forall x\ [\exists y\ Animal(y) \land \neg Loves(x, y)] \lor [\exists y\ Loves(y, x)]$.

Notice how a universal quantifier (∀ y) in the premise of the implication has become an existential quantifier. Clearly, the meaning of the original sentence has been preserved.

• **Standardize variables**: For sentences like (∃xP(x))∨(∃xQ(x)) which use the same variable name twice, change the name of one of the variables. This avoids confusion later when the quantifiers are dropped.

∀ x [∃ y Animal (y) ∧ ¬ Loves(x, y)] ∨ [∃ z Loves(z, x)] .

• **Skolemize**: **Skolemization** is the process of removing existential quantifiers by elimination. Translate ∃x P(x) into P(A), where A is a new constant.

∀ x [Animal (A) ∧ ¬ Loves(x,A)] ∨ Loves(B, x) ,

∀ x [Animal (F(x)) ∧ ¬ Loves(x, F(x))] ∨ Loves(G(z), x) .

Here F and G are **Skolem functions**. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears.

Drop universal quantifiers: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left.Therefore drop the universal quantifiers:

[Animal (F(x)) ∧ ¬ Loves(x, F(x))] ∨ Loves(G(z), x) .

• **Distribute ∨ over ∧**:

[Animal (F(x)) ∨ Loves(G(z), x)] ∧ [¬ Loves(x, F(x)) ∨ Loves(G(z), x)] .

This step may also require flattening out nested conjunctions and disjunctions. The sentence is now in CNF and consists of two clauses.

Example1: Build the given sentences to wff clause and prove the statement to "Did curiosity kill the cat?"

1.Anyone who kills an animal is loved by no one.
2. Everyone who loves all animals is loved by someone.
3. Jack loves all animals.
4. Either jack or curiosity killed the cat,who is named tuna

First, the original sentences are expressed with some background knowledge, and the negated goal G in first-order logic:

A. ∀x [∀y Animal(y) ⇒ Loves(x,y)] ⇒ [∃y Loves(y,x)]

B. ∀x [∃z Animal(z) ∧ Kills(x,z)] ⇒ [∀y ¬ Loves(y,x)]

C. ∀x Animal(x) ⇒ Loves(Jack,x)
D. Kills(Jack,Tuna) ∨ Kills(Curiosity,Tuna)
E. Cat(Tuna)
F. ∀x Cat(x) ⇒ Animal(x)

¬ G. ¬ Kills(Curiosity,Tuna)

Now the conversion procedure is applied to convert each sentence to CNF:
A1. Animal(F(x)) ∨ Loves(G(x),x)

A2. ¬ Loves(x,F(x)) ∨ Loves(G(x),x)

B. ¬ Loves(y,x) ∨ ¬ Animal(z) ∨ ¬ Kills(x,z)

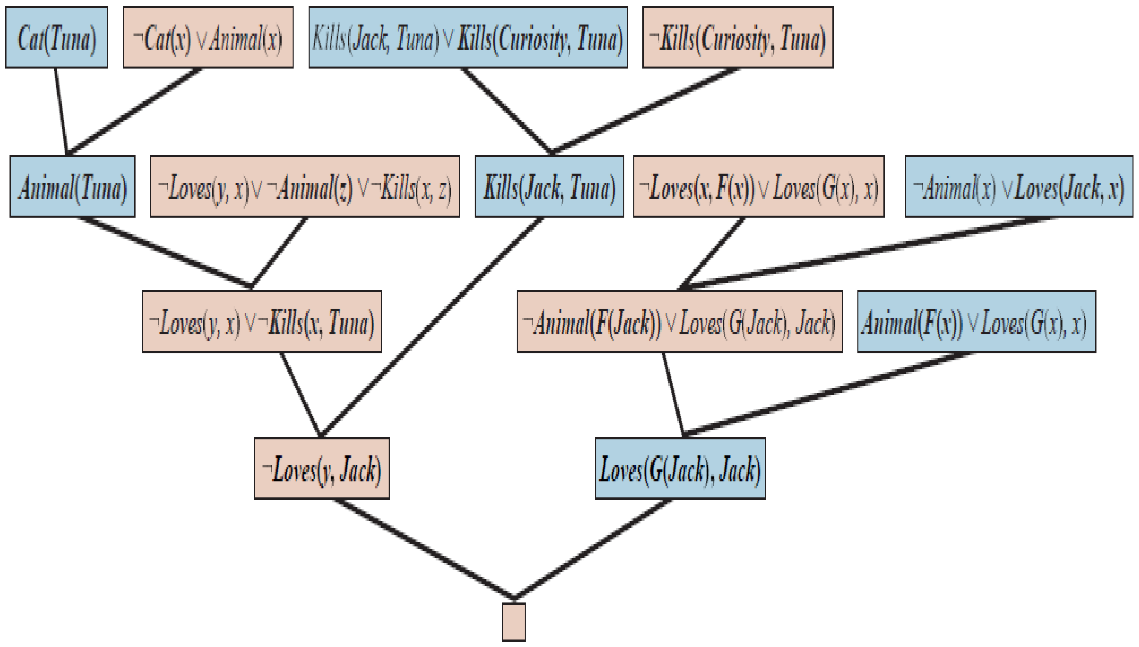
C. ¬ Animal(x) ∨ Loves(Jack,x)

D. Kills(Jack,Tuna) ∨ Kills(Curiosity,Tuna)
E. Cat(Tuna)

F. ¬ Cat(x) ∨ Animal(x)

¬ G. ¬ Kills(Curiosity,Tuna)

The resolution proof that Curiosity killed the cat is given in Figure .


Cat(Tuna)
¬Cat(x) ∨ Animal(x)
Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
¬Kills(Curiosity, Tuna)
Animal(Tuna)
¬Loves(y, x) ∨ ¬Animal(z) ∨ ¬Kills(x, z)
Kills(Jack, Tuna)
¬Loves(x, F(x)) ∨ Loves(G(x), x)
¬Animal(x) ∨ Loves(Jack, x)
¬Loves(y, x) ∨ ¬Kills(x, Tuna)
¬Animal(F(Jack)) ∨ Loves(G(Jack), Jack)
Animal(F(x)) ∨ Loves(G(x), x)
¬Loves(y, Jack)
Loves(G(Jack), Jack)

EXAMPLE 2:

Prove that "John likes peanuts" using resolution

a. John likes all kind of food.
b. Apple and vegetable are food
c. Anything anyone eats and not killed is food.
d. Bill eats peanuts and still alive
e. Sue eats everything that Bill eats.

Step-1: Conversion of Facts into FOL
In the first step we will convert all the given statements into its first order logic.

1)∀x: food(x) → likes(john, x)
2)food(apple) ∧ food(vegetable)
3)∀x:(∃y: eats(y, x) ∧ ¬ killedby(y, x)) → food(x)
4)eats(Bill, peanuts) ∧ alive(Bill)
5)∀x:eats(Bill, x)→ eats(Sue, x)
6. ∀x: ¬ killedby (x) → alive(x)**(Added predicate)**
7. ∀x: alive (x) → ¬ killedby(x) **(Added predicate)**

Conclusion:likes(John,peanuts)

Step-2: Conversion of FOL into CNF In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.

o Eliminate all implication (→) and rewrite

a. ∀x ¬ food(x) V likes(John, x)
b. food(Apple) Λ food(vegetables)
c. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
d. eats (Bill, Peanuts) Λ alive(Bill)
e. ∀x ¬ eats(Bill, x) V eats(Sue, x)
f. ∀x¬ [¬ killed(x)] V alive(x)
g. ∀x ¬ alive(x) V ¬ killed(x)
h. likes (John, Peanuts).

o Move negation (¬)inwards and rewrite .

∀x ¬ food(x) V likes(John, x)
a. food(Apple) Λ food(vegetables)
b. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
c. eats (Bill, Peanuts) Λ alive(Bill)
d. ∀x ¬ eats(Bill, x) V eats(Sue, x)
e. ∀x ¬killed(x)] V alive(x)
f. ∀x ¬ alive(x) V ¬ killed(x)
g. Likes (John, Peanuts).

o Rename variables or standardize variables

a. ∀x ¬ food(x) V likes(John, x)
b. food(Apple) Λ food(vegetables)
c. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
d. eats (Bill, Peanuts) Λ alive(Bill)
e. ∀w¬ eats(Bill, w) V eats(Sue, w)
f. ∀g ¬killed(g)] V alive(g)
g. ∀k ¬ alive(k) V ¬ killed(k)
h. Likes (John, Peanuts).

o **Eliminate existential instantiation quantifier by elimination.**

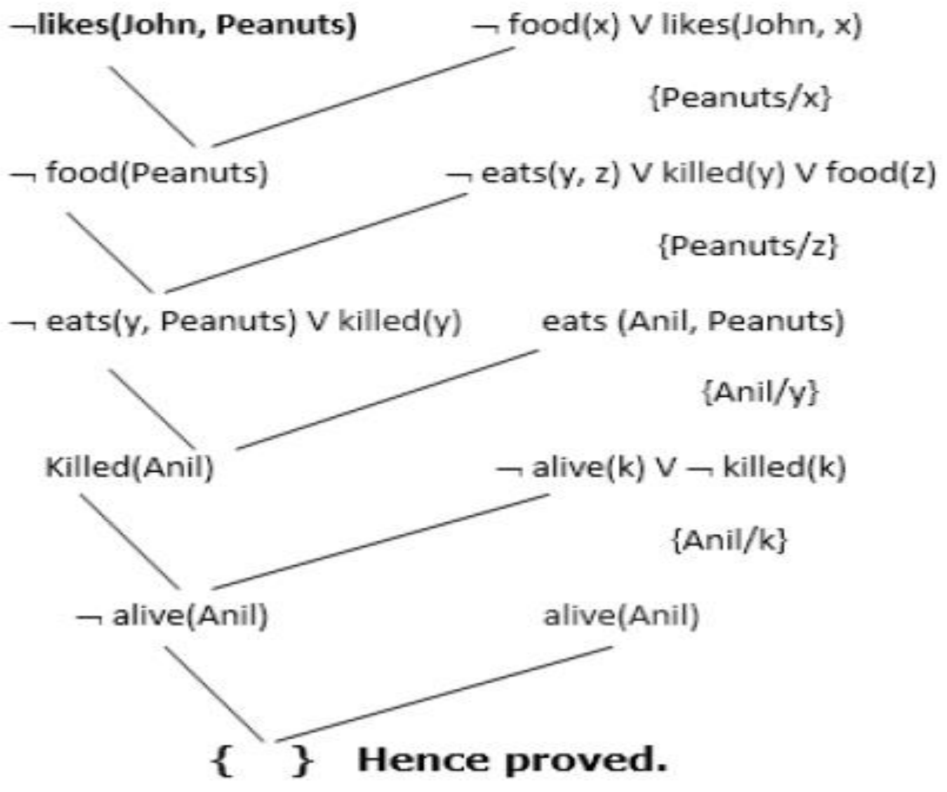
In this step, eliminate existential quantifier ∃, and this process is known as Skolemization. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.

o **Drop Universal quantifiers**. In this step drop all universal quantifier since all the statements are not implicitly quantified no need of doing this step.

a. ¬ food(x) V likes(John, x)
b. food(Apple)
c. food(vegetables)
d. ¬ eats(y, z) V killed(y) V food(z)
e. eats (Bill, Peanuts)
f. alive(Bill)
g. ¬ eats(Bill, w) V eats(Sue, w)
h. killed(g) V alive(g)
i. ¬ alive(k) V ¬ killed(k)
j. likes (John, Peanuts).

Step-3: **Negate the statement to be proved** In this statement, apply negation to the conclusion statements, which will be written as
¬likes(John, Peanuts)

Step-4: Draw Resolution graph:
Now in this step, solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:



Example 3:
1. All people who are graduating are happy.
2. All happy people smile.
3. Someone is graduating.
4. Conclusion: Is someone smiling?

Solution

Convert the sentences into predicate Logic

1. ∀x graduating(x) -> happy(x)
2. ∀x happy(x) -> smile(x)
3. ∃x graduating(x)
4. ∃x smile(x)

Convert to clausal form

(i) Eliminate the → sign

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(x) vsmile(x)
3. ∃x graduating(x)
4. -∃x smile(x)

(ii) Reduce the scope of negation

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(x) vsmile(x)
3. ∃x graduating(x)
4. ∀x¬ smile(x)

(iii) Standardize variable apart

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(y) vsmile(y)
3. ∃x graduating(z)
4. ∀x¬ smile(w)

(iv) Move all quantifiers to the left

1. ∀x¬ graduating(x) vhappy(x)
2. ∀x¬ happy(y) vsmile(y)
3. ∃x graduating(z)
4. ∀w¬ smile(w)

(v) Eliminate ∃

1. ∀x¬ graduating(x) vhappy(x)
2. ∀x¬ happy(y) vsmile(y)

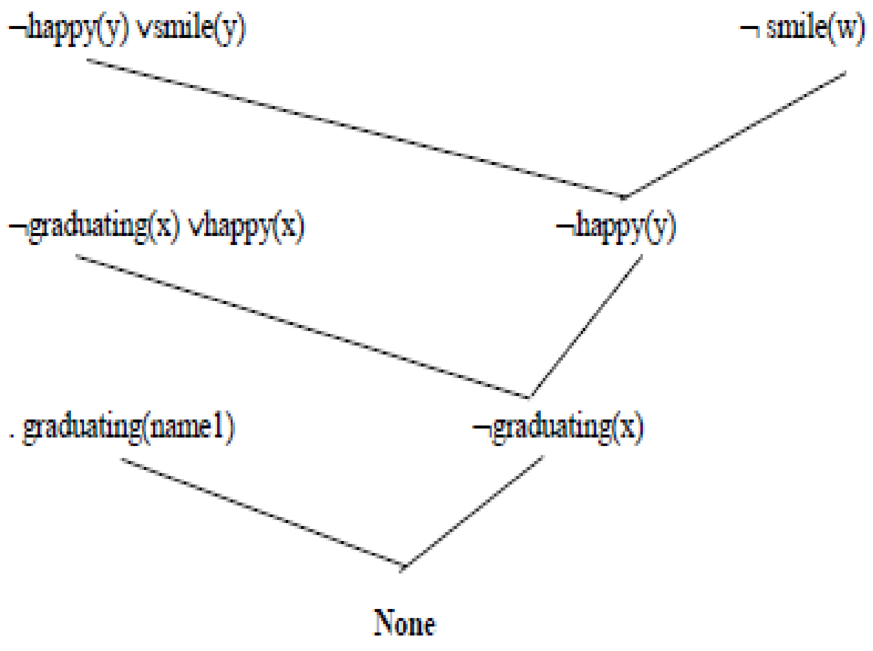
(vi) Eliminate∀

1. ¬graduating(x) vhappy(x)
2. ¬happy(y) vsmile(y)
3. graduating(name1)
4. ¬smile(w)

(vii) Convert to conjunct of disjuncts form

(viii) Make each conjunct a separate clause.

(ix) Standardize variables apart again.



Thus, we proved someone is smiling.

EXAMPLE 4

Explain the unification algorithm used for reasoning under predicate logic with an example. Consider the following facts

a. Team India

b. Team Australia

c. Final match between India and Australia

d. India scored 350 runs, Australia scored 350 runs, India lost 5 wickets, Australia lost 7 wickets.

f. If the scores are same the team which lost minimum wickets wins the match.

Represent the facts in predicate, convert to clause form and prove by resolution “India wins the match”.

Solution

Convert into predicate Logic

(a) team(India)

(b) team(Australia)

(c) team(India) ^ team(Australia) → final_match(India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x team(x) ^ wins(x) → score(x,mat_runs)

(f) ∃xy score(x,equal(y)) ^ wicket(x,min) ^ final_match(x,y) → win(x)

Convert to clausal form

(i) Eliminate the → sign

(a) team(India)

(b) team(Australia)

(c) ¬(team(India) ^ team(Australia) v final_match(India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x¬ (team(x) ^ wins(x)) vscore(x,max_runs))

(f) ∃xy¬ (score(x,equal(y)) ^ wicket(x,min) ^final_match(x,y)) vwin(x)

(ii) Reduce the scope of negation

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match(India, Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x¬ team(x) v ¬wins(x) vscore(x,max_runs))

(f) ∃xy¬ (score(x,equal(y)) v ¬ wicket(x,min_wicket) v¬final_match(x,y)) vwin(x)

(iii) Standardize variables apart

(iv) Move all quantifiers to the left

(v) Eliminate ∃

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match (India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ¬team(x) v ¬wins(x) vscore(x, max_runs))

(f) ¬score(x,equal(y)) v¬wicket(x,min_wicket) v-final_match(x,y)) vwin(x)

(vi) Eliminate∀

(vii) Convert to conjunct of disjuncts form.

(viii) Make each conjunct a separate clause.

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match (India,Australia)

(d) score(India,350)

Score(Australia,350)

Wicket(India,5)

Wicket(Austrialia,7)

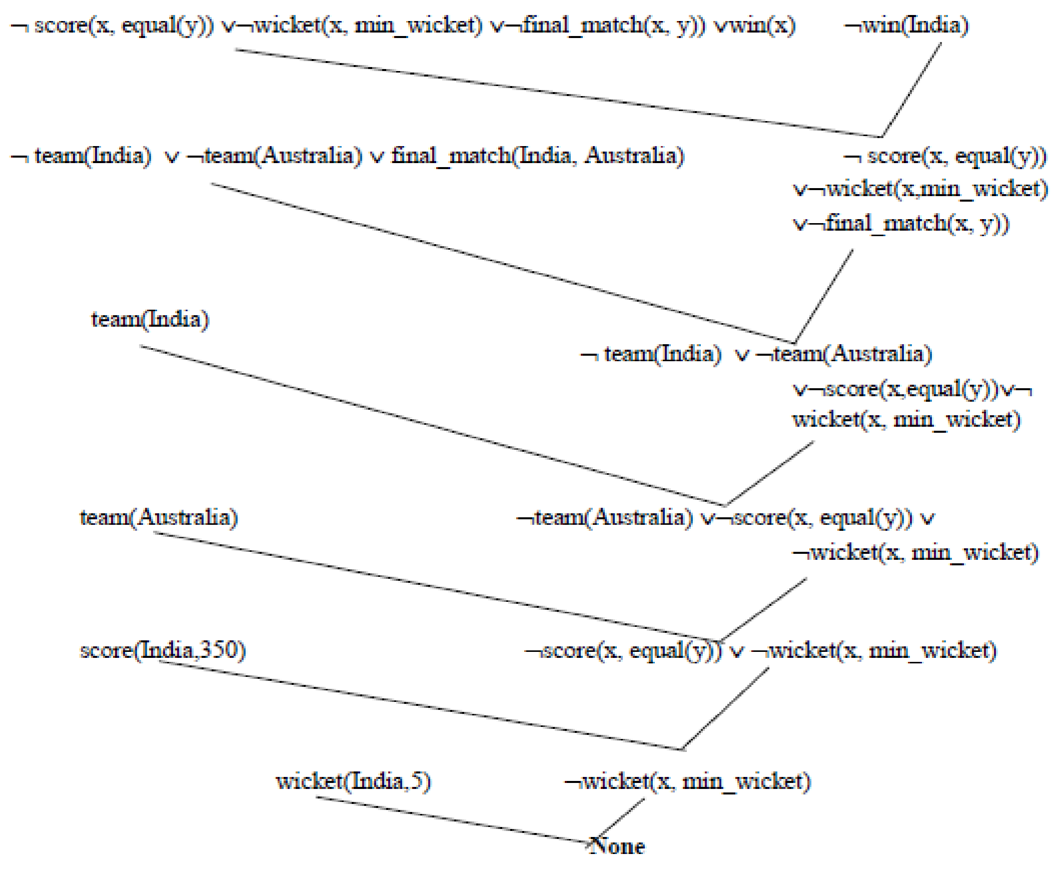
(e) ¬team(x) v ¬wins(x) vscore(x,max_runs))

(f) ¬score(x,equal(y)) v¬wicket(x,min_wicket) v¬final_match(x,y)) vwin(x)

(ix) Standardize variables apart again

To prove:win(India)

Disprove:¬win(India)


¬ score(x, equal(y)) v¬wicket(x, min_wicket) v¬final_match(x, y)) vwin(x)
¬win(India)
¬ team(India) v ¬team(Australia) v final_match(India, Australia)
¬ score(x, equal(y)) v¬wicket(x,min_wicket) v¬final_match(x, y))
team(India)
¬ team(India) v ¬team(Australia) v¬score(x,equal(y))v¬ wicket(x, min_wicket)
team(Australia)
¬team(Australia) v¬score(x, equal(y)) v ¬wicket(x, min_wicket)
score(India,350)
¬score(x, equal(y)) v ¬wicket(x, min_wicket)
wicket(India,5)
¬wicket(x, min_wicket)
None

Thus, proved India wins match.

EXAMPLE 5

Consider the following facts and represent them in predicate form:
F1. There are 500 employees in ABC company.
F2. Employees earning more than Rs. 5000 per tax.
F3. John is a manger in ABC company.
F4. Manger earns Rs. 10,000.
Convert the facts in predicate form to clauses and then prove by resolution: “John pays tax”.

Solution

Convert into predicate Logic

1. company(ABC) ^employee(500,ABC)
2. ∃x company(ABC) ^employee(x,ABC) ^ earns(x,5000)→pays(x,tax)
3. manager(John,ABC)
4. ∃x manager(x, ABC)→earns(x,10000)

Convert to clausal form

(i) Eliminate the → sign

1. company(ABC) ^employee(500,ABC)
2. ∃x¬(company(ABC) ^employee(x,ABC) ^ earns(x,5000)) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

(ii) Reduce the scope of negation

1. company(ABC) ^employee(500,ABC)
2. ∃x¬ company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

(iii) Standardize variables apart

1. company(ABC) ^employee(500,ABC)
2. ∃x¬ company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

(iv) Move all quantifiers to the left

(v) Eliminate ∃

1. company(ABC) ^employee(500,ABC)
2. ¬company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ¬manager(x, ABC) v earns(x,10000)

(vi) Eliminate∀

(vii) Convert to conjunct of disjuncts form

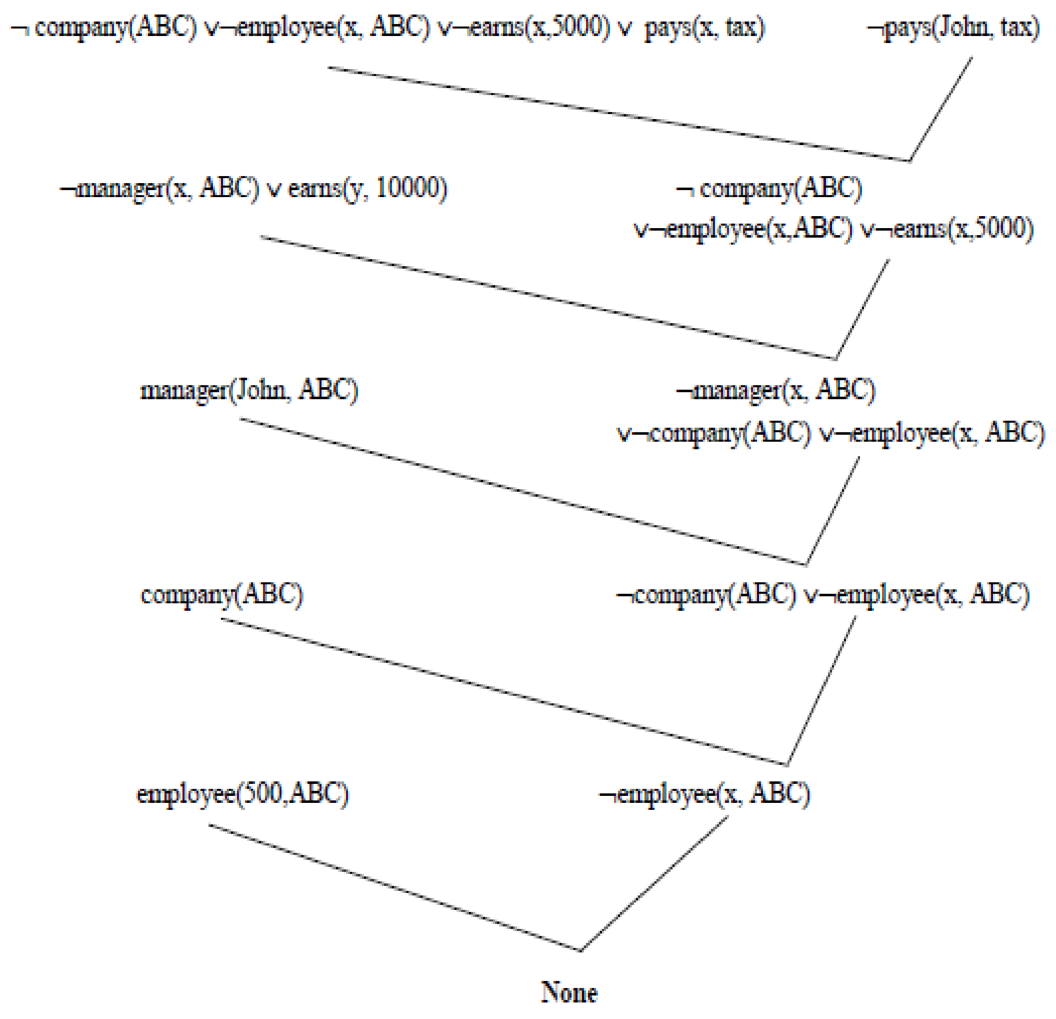
(viii) Make each conjunct a separate clause.

1. (a) company(ABC)
   (b) employee(500,ABC)
2. ¬company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ¬manager(x, ABC) v earns(x,10000)

(ix) Standardize variables apart again.

Prove:pays(John,tax)

Disprove:¬pays(John,tax)


¬ company(ABC) v¬employee(x, ABC) v¬earns(x,5000) v pays(x, tax)
¬pays(John, tax)
¬manager(x, ABC) v earns(y, 10000)
¬ company(ABC) v¬employee(x,ABC) v¬earns(x,5000)
manager(John, ABC)
¬manager(x, ABC) v¬company(ABC) v¬employee(x, ABC)
company(ABC)
¬company(ABC) v¬employee(x, ABC)
employee(500,ABC)
¬employee(x, ABC)
None

Thus, proved john pays tax.