UNIT - III: Logic Concepts

Syllabus:

UNIT - III: Logic Concepts

Introduction, propositional calculus, proportional logic, natural deduction system, axiomatic system, semantic tableau system in proportional logic, resolution in proportional logic, resolution in predicate logic and unification algorithm.

Outcomes:

Student will be able to:

3.1 Introduction

- One particular way of representing facts is the language of logic.
- The logical formalism is appealing because it immediately suggests a powerful way of deriving new knowledge from old called as "mathematical deduction".
- In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known.
- The way of demonstrating the truth of an already believed proposition can be extended to include deduction as a way of deriving answers to questions and solutions to problem.
- we use the following standard logic symbols:
  - →(Implication)
  - ¬ (Not)
  - **Λ** (And)
  - **V** (Or)
  - ∃ (there exists)
  - ∀ (For All)

3.2 Propositional logic

- Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.
- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.

- Propositional logic consists of an object, relations or function, and logical connectives.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called tautology, and it is also called a valid sentence.
- A proposition formula which is always false is called Contradiction.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "Where is Rohini", "How are you", "What is your name", are not propositions.
- The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:
  - **Atomic Propositions:** Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.
    Examples:
    a) 2+2 is 4, it is an atomic proposition as it is a true fact.
    b) "The Sun is cold" is also a proposition as it is a false fact.
  - **Compound propositions:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.
    Examples:
    a) "It is raining today, and street is wet."
    b) "Ankit is a doctor, and his clinic is in Mumbai."
- **Logical connectives** are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:
  1. **Negation:** A sentence such as $\neg$ P is called negation of P. A literal can be either Positive literal or negative literal.
  2. **Conjunction:** A sentence which has $\wedge$ connective such as, P $\wedge$ Q is called a conjunction.
     Example: Rohan is intelligent and hardworking. It can be written as,
     P= Rohan is intelligent,
     Q= Rohan is hardworking. $\rightarrow$ P$\wedge$ Q.
  3. **Disjunction:** A sentence which has $\vee$ connective, such as P $\vee$ Q. is called disjunction, where P and Q are the propositions.

Example: "Ritika is a doctor or Engineer", Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as P ∨ Q.

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

➢ **Truth Tables:**

P	**Q**	**¬Q**	**P ∧ Q**	**P ∨ Q**	**P → Q**	**P⇔ Q**
T	T	F	T	T	T	T
T	F	T	F	T	F	F
F	T	F	F	T	T	F
F	F	T	F	F	T	T

➢ **Limitations of Propositional logic:**

- We cannot represent relations like ALL, some, or none with propositional logic. Example:
  All the girls are intelligent.
  Some apples are sweet.
- Propositional logic has limited expressive power.
- In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

<u>3.3 Semantic Tableau in propositional logic</u>

➢ **Semantic tableau** is designed to make semantic reasoning fully systematic and to generate decisions automatically as to whether sequents are valid or not.

➢ The idea of a tableau is to take a set of formulae with labels saying which are true and which false, and to process them by analysing them into their subformulae, transferring the labels to those in accordance with the semantics of the various connectives, until either the analysis is complete or a contradiction results.

➢ Each node of the tableau is characterised by a set of labelled formulae, and if it has successor nodes, these result by analysis of some formula in the set into its principal subformulae. For example, suppose there is a conjunction *A* ∧ *B* in the set with the label **t** (i.e. true). For a conjunction to be true is exactly for both of its conjuncts to be true, so we can choose to extend the tableau by analysing the conjunction, in which case the successor node will have the two formulae *A* and *B* each with label **t** in place of the conjunction. On the other hand, if we choose to analyse a disjunction *A* ∨ *B* also with the label **t**, there will be no unique successor node, since there is nothing to say which disjunct should be the true one. Instead, there will be two successors, one with *A* labelled **t** in place of the disjunction, and the other with *B*.

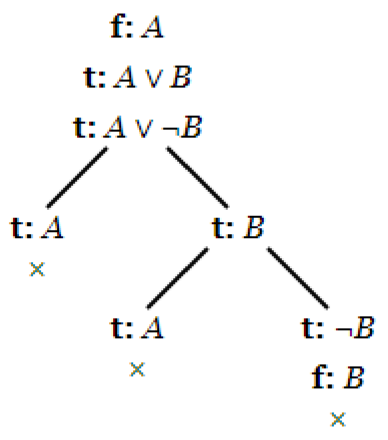
- The rules for extending tableaux to analyse negations, conjunctions and disjunctions:

$$\frac{\mathbf{t}: \neg A}{\mathbf{f}: A} \qquad \frac{\mathbf{t}: A \land B}{\mathbf{t}: A,\ \mathbf{t}: B} \qquad \frac{\mathbf{t}: A \lor B}{\mathbf{t}: A \mid \mathbf{t}: B} \qquad \frac{\mathbf{t}: A \to B}{\mathbf{f}: A \mid \mathbf{t}: B} \qquad \frac{\mathbf{t}: A \leftrightarrow B}{\mathbf{t}: A,\ \mathbf{t}: B \mid \mathbf{f}: A,\ \mathbf{f}: B}$$

$$\frac{\mathbf{f}: \neg A}{\mathbf{t}: A} \qquad \frac{\mathbf{f}: A \land B}{\mathbf{f}: A \mid \mathbf{f}: B} \qquad \frac{\mathbf{f}: A \lor B}{\mathbf{f}: A,\ \mathbf{f}: B} \qquad \frac{\mathbf{f}: A \to B}{\mathbf{t}: A,\ \mathbf{f}: B} \qquad \frac{\mathbf{f}: A \leftrightarrow B}{\mathbf{t}: A,\ \mathbf{f}: B \mid \mathbf{f}: A,\ \mathbf{t}: B}$$

A set of labelled formulae including the one above the line may be developed by replacing that formula with the one or more formulae below the line. A vertical bar separating two sets below the line, indicates that there are two successor nodes, so the branch of the tableau splits into two at that point.

- A contradiction in a tableau branch causes that branch to die: it gets closed off and no more growth happens from its tip. If every branch dies, the tree dies; in that case we know there was no way the tested formulas could all have had the truth values they were initially supposed to have.
- If, on the other hand, we get to a state in which every subformula has been processed in every branch in which it occurs and some branch is left alive or "open", we have found an interpretation on which all of the original formulas are true. Conventionally we mark a dead or "closed" branch by writing a cross under it.



3.4 Representing Simple Facts in Logic

- Propositional logic is appealing because it is simple to deal with and a decision procedure for it exists.
- We can easily represent real-world facts as logical propositions written as well formed formulas (wff's) in Propositional logic.
- Some Simple Facts in Propositional logic:
  - It is raining
    RAINING
  - It is sunny
    SUNNY
  - It is windy
    WINDY
  - If it is raining, then it is not sunny.
    RAINING→¬SUNNY
- Suppose we want to represent the fact stated by the sentence:
  - Socrates is a man
    We could write SOCRATESMAN. It is represented as:
    MAN (SOCRATES)
  - Plato is a man
    MAN (PLATO)
- Now the structure of the representation reflects the structure of the knowledge itself. But to do that, we need to be able to use "**predicates"** applied to arguments.
- To represent the following sentence:
  All men are mortal

We can't represent this as: **MORTALMAN**

But that fails to capture the relationship between any individual being a man and that individual being a mortal. To do that, we need **variables and quantification.**

- So we use **first-order predicate logic** as a way of representing knowledge because it permits representations of things that cannot be represented in propositional logic.
- In predicate logic, we can represent real-world facts as statements written as wff's.
- The major motivation for choosing to use logic is that it we use logical statements as a way of representing knowledge, and then we have a good way of reasoning with that knowledge.
- Determining the validity of a proposition in propositional logic is straightforward, although it may be computationally hard.

- Before we use predicate logic as a medium for representing knowledge, we need to check whether it also provides a good way of reasoning with the knowledge.
  - It provides a way of deducing new statements from old ones.
  - Unfortunately, unlike propositional logic, it does not possess a decision procedure.
  - There do exist procedures that will find a proof of a proposed theorem, but first-order predicate logic is not decidable, it is semi-decidable.
  - A simple such procedure is to use the rules of inference to generate theorems from the axioms in some orderly fashion. This method is not particularly efficient, however, and we will want to try to find a better one.

Use of predicate logic:

- Let's now explore the use of predicate logic as a way of representing knowledge by looking at a following example:

  Consider the following set of sentences:
  1. Marcus was a man.
  2. Marcus was a Pompeian.
  3. All Pompeians were Romans.
  4. Caesar was a ruler.
  S, All Romans were either loyal to Caesar or hated him.
  6. Everyone is loyal to someone.
  7. People only try to assassinate rulers they are not loyal to.
  8. Marcus tried to assassinate Caesar.

- The facts described by these sentences can be represented as a set of wff's in predicate logic as follows:

  1. Marcus was a man.

  man (Marcus)

  This representation captures the critical fact of Marcus being a man. It fails to capture some of the information in the English sentence, the notion of past tense.

  2. Marcus was a Pompeian.

  Pompeian (Marcus)

  3. All Pompeians were Romans.

  ∀(x): **Pompeian(x) → Roman(x)**

  4. Caesar was a 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): 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)

- Now suppose that we want to use these statements to answer the question:

  Was Marcus loyal to Caesar?

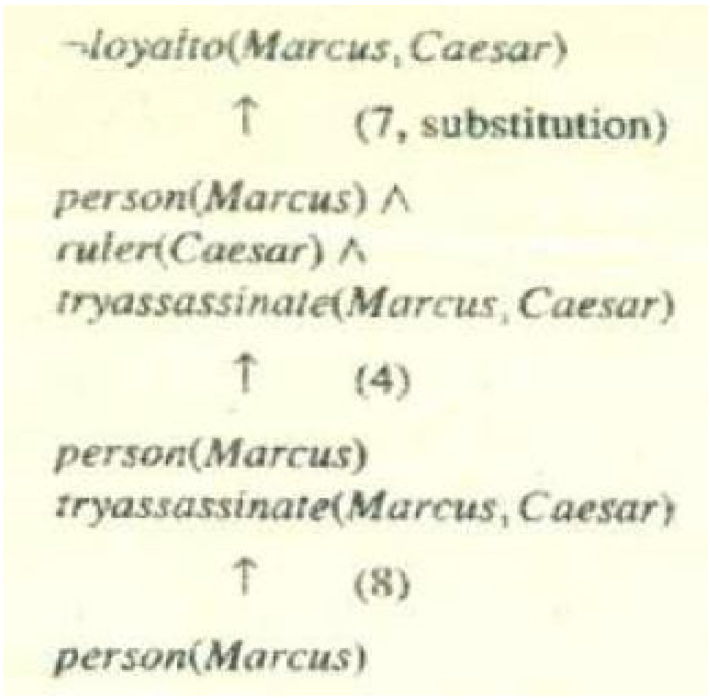
  Let's try to produce a formal proof, reasoning backward from the desired goal:

  ¬loyalto (Marcus, Caesar)

  To prove this, let us add a fact:

  All men are people

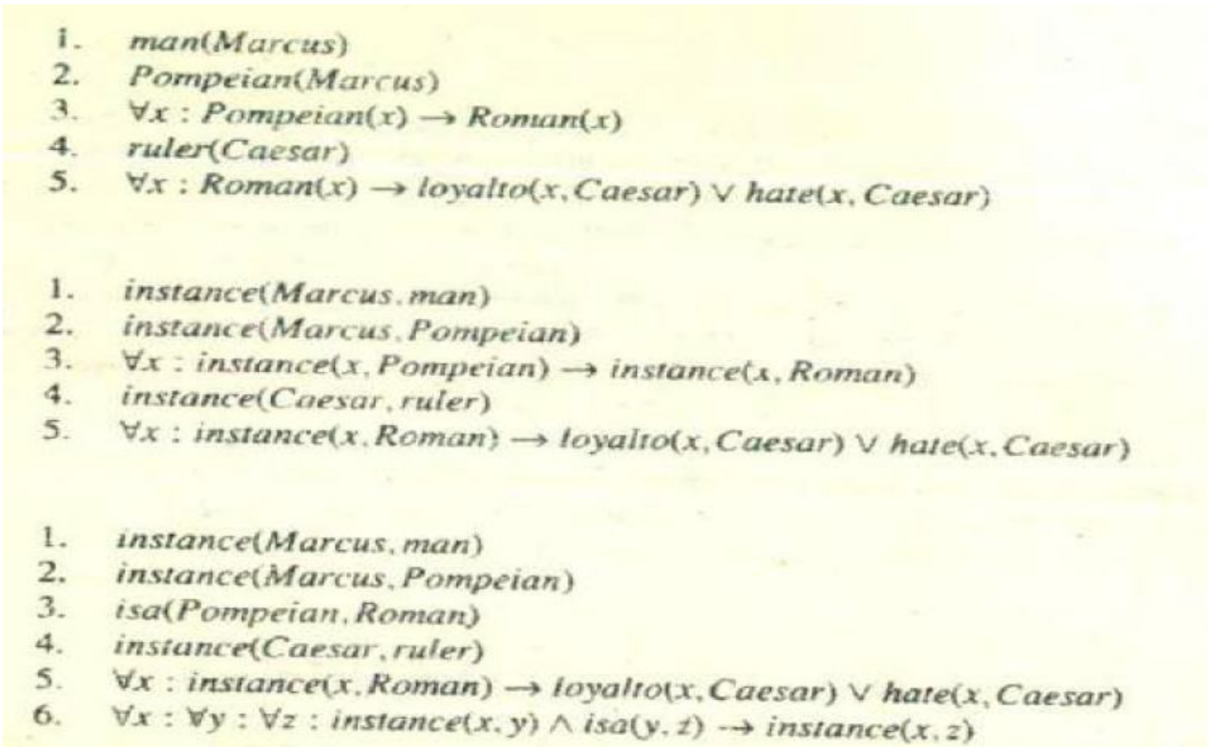
  Although we know that Marcus was a man, we do not have any way to conclude that Marcus was a person. So, we need the representation of another fact to our system, **∀(x): man(x)→person(x)**



3.5 Representing Instance and isa Relationships

- The specific attributes **instance and isa** play an important role in the useful form of reasoning, **property inheritance**.
- Let us represent the following facts in three ways:
  1. Marcus was a man.
  2. Marcus was a Pompeian.
  3. All Pompeians were Romans.

4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.



1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\forall x : Pompeian(x) \rightarrow Roman(x)$
4. $ruler(Caesar)$
5. $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $\forall x : instance(x, Pompeian) \rightarrow instance(x, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $isa(Pompeian, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$
6. $\forall x : \forall y : \forall z : instance(x, y) \wedge isa(y, z) \rightarrow instance(x, z)$

- Second representation: The predicate ***instance*** *is* a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit ***isa*** predicate.
- In subclass relationships, such as that between Pompeians and Romans, the implication rule there states that it an object is an instance of the subclass *Pompeian* then it is an instance of the superclass *Roman.*
- This rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship.
- Third representation: It contains representations that use both the ***instance* and *isa* predicates** explicitly.
- The use of the *isa* predicate simplifies the representation of sentence 3, but it requires that one additional axiom. This additional axiom describes how an *instance* relation and an *isa* relation can be combined to derive a new *instance* relation.

3.6 Representing facts in Propositional Logic:

Consider the following set of facts:
1. Marcus was a man.
2. Marcus was a Pompeian.

3. Marcus was boni in 40 A.D.
4. All men are mortal.
5. All Pompeians died when the volcano erupted in 79 AD.
6. No mortal lives longer than 150 years.
7. It is now 1991.
8. Alive means not dead.
9. If someone dies, then he is dead at all later times.

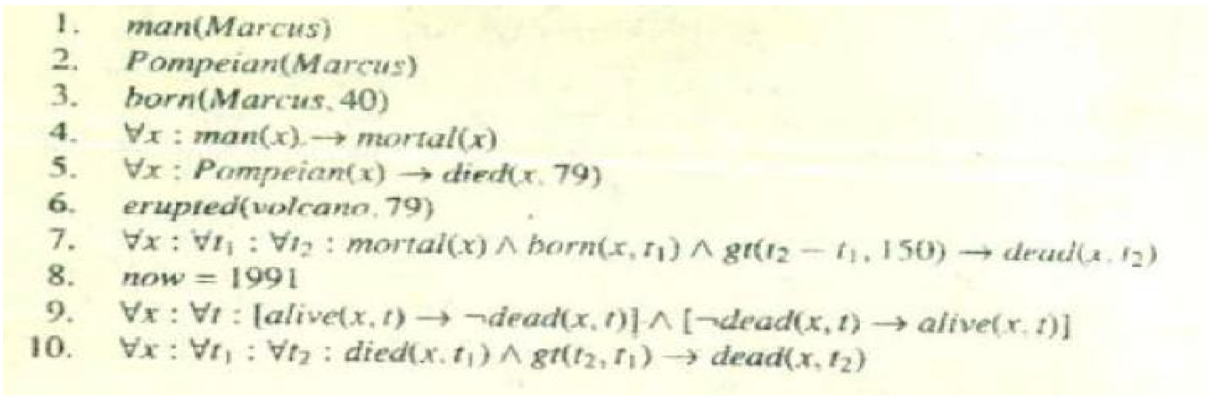
Now let's attempt to answer the question:

"Is Marcus alive?"

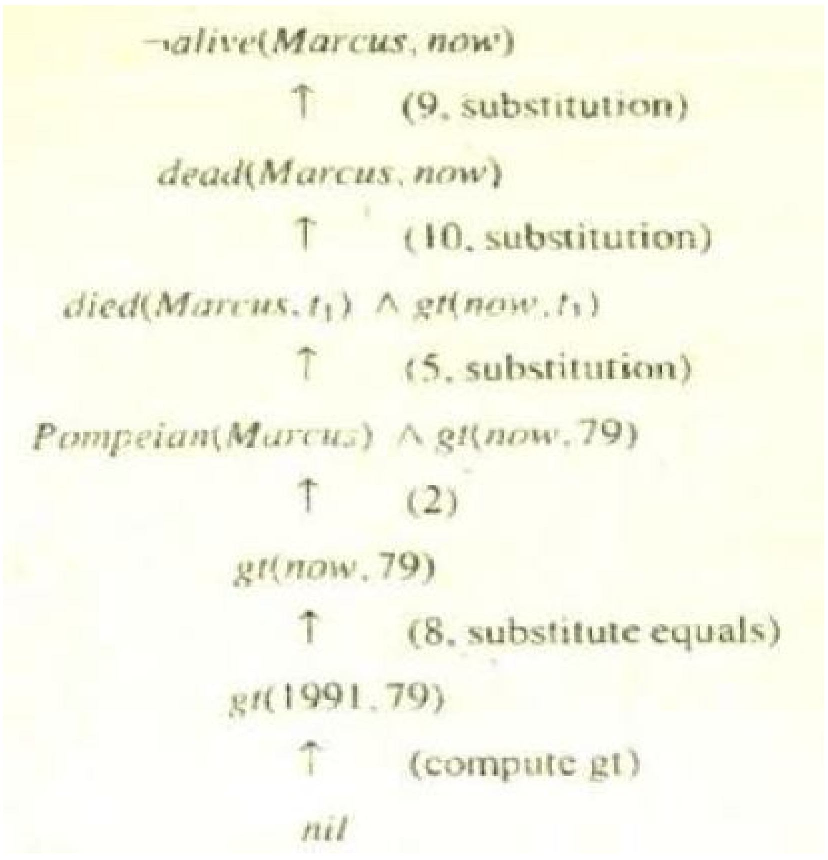
By proving:

¬ alive(Marcus, now)

The given facts are represented as follows:



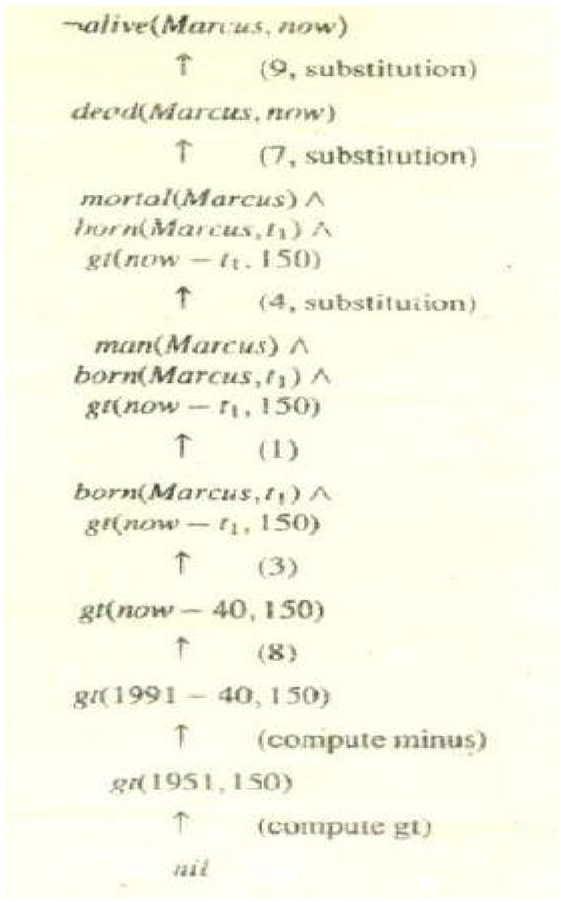
1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\forall x : man(x) \rightarrow mortal(x)$
5. $\forall x : Pompeian(x) \rightarrow died(x, 79)$
6. $erupted(volcano, 79)$
7. $\forall x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$
8. $now = 1991$
9. $\forall x : \forall t : [alive(x, t) \rightarrow \neg dead(x, t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$
10. $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$



$$\begin{tikzcd}
\neg alive(Marcus, now) \\
dead(Marcus, now) \arrow[u, "\text{(9, substitution)}"'] \\
died(Marcus, t_1) \wedge gt(now, t_1) \arrow[u, "\text{(10, substitution)}"'] \\
Pompeian(Marcus) \wedge gt(now, 79) \arrow[u, "\text{(5, substitution)}"'] \\
gt(now, 79) \arrow[u, "\text{(2)}"'] \\
gt(1991, 79) \arrow[u, "\text{(8, substitute equals)}"'] \\
nil \arrow[u, "\text{(compute gt)}"']
\end{tikzcd}$$

3.7 Resolution:

- Resolution is such a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form.
- Resolution produces proofs by ***refutation***. To prove a statement (to show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e.. that it is unsatisfiable).
- Resolution is a technique for proving theorems in the propositional or predicate calculus that has been a part of AI problem-solving.
- Resolution describes a way of finding contradictions in a database of clauses with minimum use of substitution. Resolution refutation proves a theorem by negating the statement to be proved and adding this negated goal to the set of axioms that are known (have been assumed) to be true.
- It then uses the resolution rule of inference to show that this leads to a contradiction. Once the theorem prover shows that the negated goal is inconsistent with the given set of axioms, it follows that the original goal must be consistent. This proves the theorem.



$\neg alive(Marcus, now)$

$\uparrow$ (9, substitution)

$dead(Marcus, now)$

$\uparrow$ (7, substitution)

$mortal(Marcus) \wedge born(Marcus, t_1) \wedge gt(now - t_1, 150)$

$\uparrow$ (4, substitution)

$man(Marcus) \wedge born(Marcus, t_1) \wedge gt(now - t_1, 150)$

$\uparrow$ (1)

$born(Marcus, t_1) \wedge gt(now - t_1, 150)$

$\uparrow$ (3)

$gt(now - 40, 150)$

$\uparrow$ (8)

$gt(1991 - 40, 150)$

$\uparrow$ (compute minus)

$gt(1951, 150)$

$\uparrow$ (compute gt)

nil

- Resolution refutation proofs involve the following steps:
  1. Put the premises or axioms into clause form.
  2. Add the negation of what is to be proved, in clause form, to the set of axioms.
  3. Resolve these clauses together, producing new clauses that logically follow from them.
  4. Produce a contradiction by generating the empty clause.
  5. The substitutions used to produce the empty clause are those under which the opposite of the negated goal is true.

3.8 Conversion to Clause Form:

- Consider the following fact:

  All Romans who know Marcus either hate Caesar or think that anyone who hates anyone is crazy.

  ∀x:[Roman(X)∧know(x,Marcus)]→[hate(x,Caesar)V(∀y:∃z:hate(y, z)→thinkcrazy(x, y))]

Algorithm: Conversion to clause form:

1. Eliminate→, using the fact that **a→b** is equivalent to**¬ aVb**. Performing this transformation, we get:

   ∀x:**¬**[Roman(X)∧know(x,Marcus)]**V**[hate(x,Caesar)V(∀y:**¬**(∃z:hate(y, z))**V**thinkcrazy(x, y))]

2. Reduce the scope of each **¬** to a single term, using the fact that ¬(¬P)=P and deMorgan's laws [which say that (¬(a∧b)= ¬aV ¬b and (¬(a V b)= ¬a∧ ¬b) and the standard correspondences between quantifiers:- ∀x: P(x)= ∃x: ¬P(x) and ∃x: P(x)= ∀x: ¬P(x) Performing this , we get:

   ∀x:[**¬**Roman(X)**V¬**know(x,Marcus)]**V**[hate(x,Caesar)V(∀y:∀z:**¬**hate(y, z))V(thinkcrazy(x, y))]

3. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff. For example, the formula:

   ∀x : P(x) V ∀x: Q(x) would be converted to:

   ∀x : P(x) V ∀y: Q(y)

4. Move all quantifiers to the left of the formula without changing their relative order. This is possible since there is no conflict among variable names. Performing this, we get:

   ∀x:∀y:∀z:[**¬**Roman(X)**V¬**know(x,Marcus)]**V**[hate(x,Caesar)V(**¬**hate(y, z)**V**thinkcrazy(x, y))]

At this point, the formula is known as **Prenex normal form**. It consists of a prefix of quantifiers followed by a matrix, which is quantifier-free.

5. Eliminate existential quantifiers.
For example, the formula
∃y:President(y) can be transformed into the formula:
President(S1) where S1 is a function with no arguments that somehow produces a value that satisfies **President.**
If existential quantifiers occur within the scope of universal quantifiers, then the value that satisfies the predicate may depend on the values of the universally quantified variables. For example:

∀x: ∃y: father-of(y, x)

The value of' y that satisfies father-of depends on the particular value of x, we must generate functions with the same number of arguments as the number of universal quantifiers in whose scope the expression occurs. So this would be transformed into:

∀x: father-of(S2(x), x))

These generated functions are called **Skolem Functions**. Sometimes ones with no arguments are called **Skomlem constants**.
6. Drop the prefix. All remaining variables are universally quantifled, so the prefix can just be dropped and any proof procedure we use can simply assume that any variable it sees is universally quantified.

[**¬**Roman(X)**V¬**know(x,Marcus)]**V**[hate(x,Caesar)V
(**¬**hate(y, z)**V**thinkcrazy(x, y))]

7. Convert the matrix into a conjunction of disjuncts. Since there are no and's, it is only necessary to exploit the associative property of (i.e., aV(b Vc) =(a Vb)Vc and simply remove the parentheses.

¬Roman(X)**V¬**know(x,Marcus)**V**hate(x,Caesar)V (**¬**hate(y, z)
Vthinkcrazy(x, y)

8 Create a separate clause corresponding to each conjunct. In order for a wff to be true, all the clauses that are generated from it must be true. If we are going to be working with several wff's, all the clauses generated by each of them can now be combined to represent the same set of facts as were represented by the original wff's.
9. Standardize apart the variables in the set of clauses.
(∀x:P(x)⋀Q(x))=∀x:P(x)⋀ ∀x:Q(x)
Since each clause is a separate conjunct and since all the variables are universally quantified, there is no relationship between the variables of two clauses, even if they were generated from the same wff.

3.9 The Basis of Resolution

- The resolution procedure is a simple iterative process: at each step two clauses, called the parent clauses, are compared (resolved), yielding a new clause that has been inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose there are two clauses in the system:

winter V summer

¬winter V cold

- Now we observe that precisely one of winter and ¬winter will be true at any point.
- If winter is true, then cold must be true to guarantee the truth of the second clause.
- If ¬winter is true, then summer must be true to guarantee the truth of the first clause.
- Thus we see that from these two clauses we can deduce:

  summer V cold
- Resolution operates by taking two clauses that each contains the same literal, in this example, winter. The literal must occur in positive form in one clause and in negative form in the other. The resolvent is obtained by combining all of the literals of the two parent clauses except the ones that cancel.
- If the clause that is produced is the empty clause, then a contradiction has been found. For example. the two clauses:

  Winter

  ¬winter

  will produce the empty clause.
- In predicate logic, the situation is more complicated since we must consider all possible ways of substituting values for the variables.

3.10 Resolution in Propositional Logic

- In propositional logic, the procedure for producing a proof by resolution of proposition *P* with respect to a set of axioms *F is* the following

Algorithm: Propositional Resolution

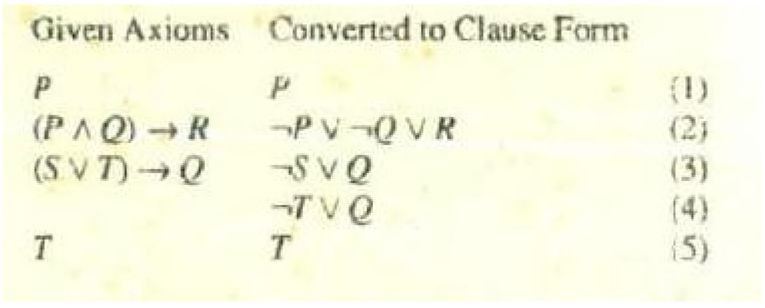
1. Convert all the propositions of *F* to clause form.
2. Negate *P* and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:

   (a) Select two clauses. Call these the parent clauses.

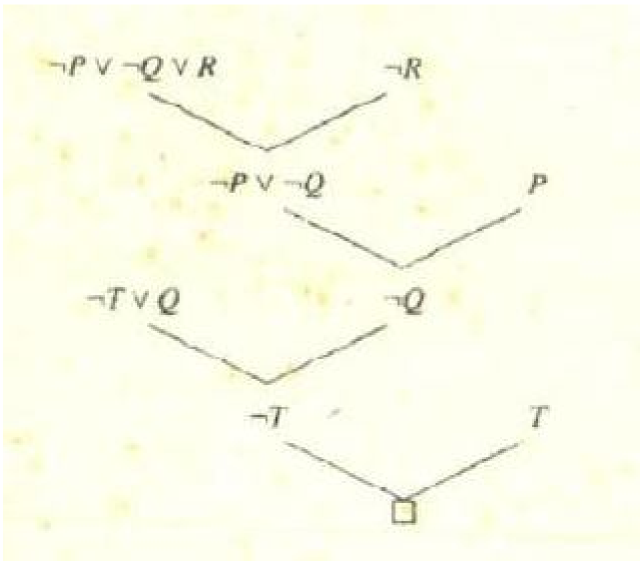
   (b) Resolve them together. The resulting clause, called the **resolvent,** will be the disjunction of all of the literals of both of the parent clauses with the following exception: It there are any pairs of literals *L* and ¬L such that one of the parent clauses contains L and the other contains ¬L ,then select one such pair and eliminate both *L* and ¬L from the resolvent.

(c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

- Suppose we are given the axioms shown below and we want to prove R. First we convert the axioms to clause form. Then we negate R. producing ¬R, which is already in clause form.
- Then we begin selecting pairs of clauses to resolve together. Although any pair of clauses can be resolved, only those pairs that contain complementary literals will produce a resolvent that is likely to lead to the goal of producing the empty clause.
- We begin by resolving with the clause ¬R .
- One way of viewing the resolution process is that it takes a set of clauses that are all assumed to be true and, based on information provided by the others, it generates new clauses that represent restrictions on the way each of those original clauses can be made true.
- A contradiction occurs when a clause when there is no way it can be true. This is indicated by the generation of the empty clause.



Given Axioms	Converted to Clause Form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)



3.11 The Unification Algorithm

- In predicate logic, this matching process is more complicated since the arguments of the predicates must be considered.
- For example, man(John) and ¬man(John) is a contradiction, while man(John) and ¬man{Spot) is not.
- Thus, in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. The straightforward recursive procedure, called the "**unification algorithm"** does this.
- **Basic idea of unification**: It is very simple. To attempt to unify two literals, we first check if their initial predicate symbols are the same. If so, we can proceed. Otherwise, there is no way they can he unified, regardless of their arguments.
- For example, the two literals:

  trytoassassinate(Marcus, Caesar)

  hate(Marcus, Caesar)

  cannot be unified.
- If the predicate symbols match, then we must check the arguments one pair at a time. If the first matches, we can continue with the second, and so on. To test each argument pair, we can simply call the unification procedure recursively.
- The matching rules are simple: Different constants or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a predicate expression with the restriction that the predicate expression must not contain any instances of the variable being matched.
- We must find a single, consistent substitution for the entire literal, not separate ones for each piece of it. To do this, we must take each substitution that we find and apply it to the remainder of the literals before we continue trying to unify them.
- For example, suppose we want to unify the expressions

  P(x, x)

  P(y, z)
- The two instances of P match. Next we compare x and y, and decide that if we substitute **y for x**, they could match. We will write that substitution as: **y/x.**
- But now, if we simply continue and match x and z, we produce the substitution **z/x.** But we cannot substitute both y and z for x. The problem can be solved as follows:

- What we need to do after finding the first substitution **y/x** is to make that substitution throughout the literals, giving:

  P(y, y)
  P(y, z)
- Now we can attempt to unify arguments y and z, which succeeds with the substitution **z/y.** The entire unification process has now succeeded with a substitution that is the composition of the two substitutions: **(z/y) (y/x).**
- In general, substitutions: **(a1/a2, a3/a4,….)(b1/b2, b3/b4….)** means to apply all the substitutions of the right most list, then take the result and apply all the ones of the next list, and so forth, until all substitutions have been applied.
- The objective of the unification procedure is to discover at least one substitution that causes two literals to match.
- For example, the literals:

  hate (x, y)
  hate (Marcus, z)

  could be unified with any of the following substitutions:

  (Marcus/x, z/y)
  (Marcus/x, y/z)
  (Marcus/x, Caesar/y, Caesar/z)
  (Marcus/x, Polonius/y, Polonius/z)
- We describe a procedure **Unify (Ll, L2),** which returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was found without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

Algorithm: Unify (L1, L2)

```
1. If L1 and L2 are both variables and constants, then:
      (a) If L1 and L2 are identical, then return NIL.
      (b) Else if L1 is a variable, then if Ll occurs in L2 then return {FAIL},
          else return (L2/L1).
      (c) Else if L2 is a variable then if L2 occurs in L1 then return {FAIL},
          else return (Ll/L2)
      (d) Else return {FAIL}.
2. If the initial predicate symbols in Ll and L2 are not identical, then return
   {FAIL}.
3. If Ll and L2 have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all
   the substitutions used to unify L1 and L2).
5. For i←1 to number of arguments in Ll:
    (a) Call Unify with the ith argument of LI and ith argument of L2,
        putting result in S.
    (b) If contains FAIL, then return {FAIL}.
    (c) If S not equal to NIL then:
        i. Apply S to the remainder of both L1 and L2.
        ii. SUBST= APPEND(S, SUBST).
6. Return SUBST.
```

3.12 Resolution in Predicate Logic

- With Unification, we now have an easy way of determining that two literals are contradictory, if one of them can be unified with the negation of the other.
- So, for example, man(x) and ¬man(Spot) are contradictory, since man(x) and man(Spot) can be unified, with substitution x/spot.
- In order to use resolution for expressions in the predicate logic, we use the unification algorithm to locate pairs of literals that cancel out.
- For example, suppose we want to resolve two clauses:

  man(Marcus)

  ¬man(x1) V mortal(x1)

  The literal man(Marcus)can be unified with the literal ¬man(x1) with the substitution **Marcus/x1.** we can now conclude only that **mortal(Marcus)** must be true.
- So the resolvent generated by clauses 1 and 2 must be **mortal (Marcus)**, which we get by applying me result of the unification process to the resolvent. The resolution process can then proceed from there to discover whether **mortal (Marcus)** leads to a contradiction with other available clauses.

Algorithm: Resolution in predicate logic

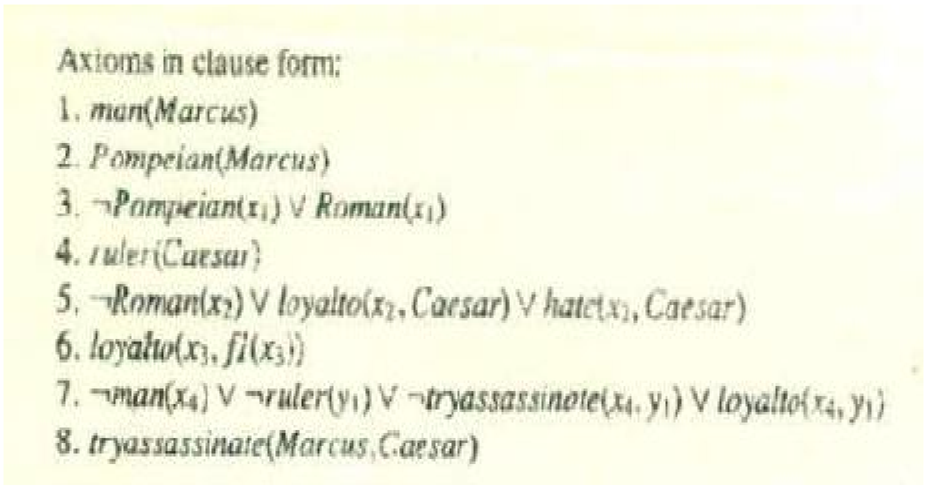
```
1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the-set of clauses
   obtained in 1.
3. Repeat until a contradiction is found, no progress can be made, or a
   predetermined amount of effort has been expended.
    (a) Select two clauses. Call these the parent clauses.
    (b) Resolve them together. The resolvent will be the disjunction of all
        the literals of both parent clauses with appropriate substitutions
        performed and with the following exception: If there is one pair of
        literals TI and ¬T2 such that one of the parent clauses contains T1
```

and the other contains ¬T2 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 **Complementary Literals**. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.

(c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of c1auses available to the procedure.

- There are systematic strategies for making the choice of clauses to resolve together each step so that we will find a contradiction if one exists. This can speed up the process considerably.
    - Only resolve pairs of clauses that contain complementary literals, since only such resolutions produce new clauses that are harder to satisfy than their parents. To facilitate this, index clauses by the predicates they contain, combined with an indication of whether the predicate is negated Then, given a particular clause, possible resolvent that contain a complementary occurrence of one of its predicates can be located directly.
    - Eliminate certain clauses as soon as they are generated so that they cannot participate in later resolutions. Two kinds of clauses should be eliminated:
        - Tautologies (which can never be unsatisfied) and
        - Clauses that are subsumed by other clauses. For example, PVQ can be subsumed by P.
    - Whenever possible, resolve either with one of- the clauses that is part of the statement we are trying to refute or with a clause generated by a resolution with such a clause. This is called the **"set-of-support strategy".**
    - Whenever possible, resolve with clauses that have a single literal. Such resolutions generate new clauses with fewer literals than the larger of their parent clauses and thus are probably closer to the goal of a resolvent with zero terms. This method is called the **"unit-preference-strategy".**

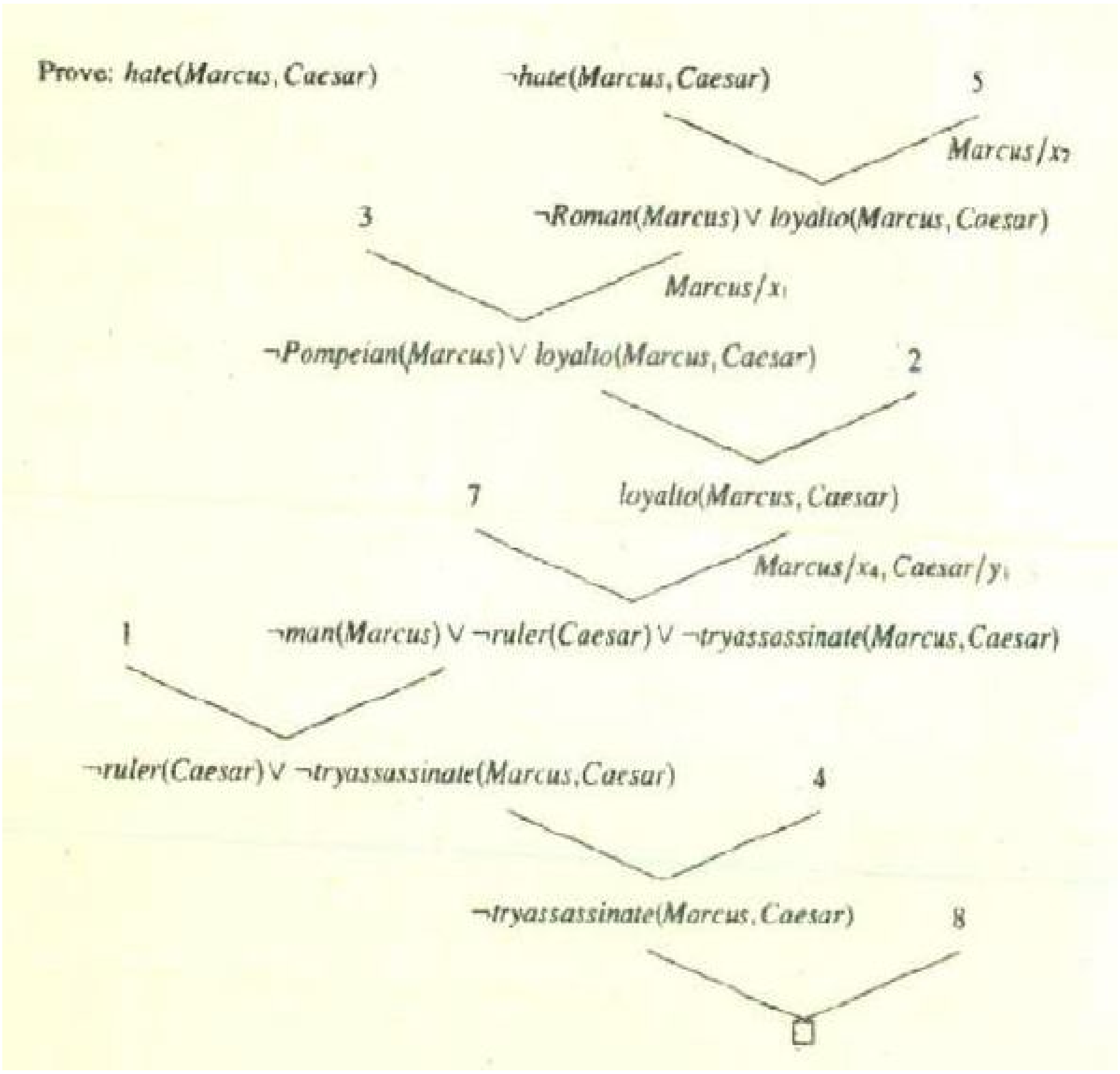
Example 1:



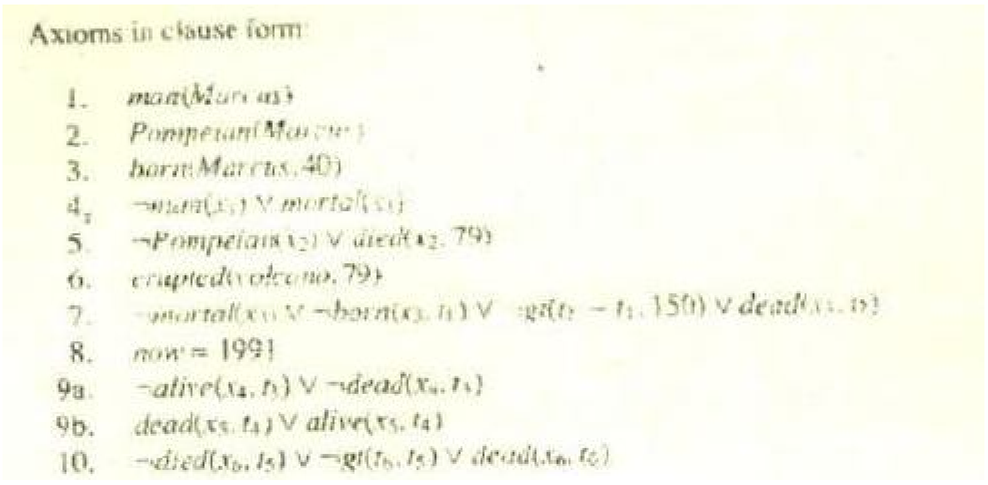
Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\neg Pompeian(x_1) \vee Roman(x_1)$
4. $ruler(Caesar)$
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$
6. $loyalto(x_3, f1(x_3))$
7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinate(x_4, y_1) \vee loyalto(x_4, y_1)$
8. $tryassassinate(Marcus, Caesar)$

Resolution Proof:


Prove: hate(Marcus, Caesar)
¬hate(Marcus, Caesar)
5
Marcus/x7
3
¬Roman(Marcus) ∨ loyalto(Marcus, Caesar)
Marcus/x1
¬Pompeian(Marcus) ∨ loyalto(Marcus, Caesar)
2
7
loyalto(Marcus, Caesar)
Marcus/x4, Caesar/y1
1
¬man(Marcus) ∨ ¬ruler(Caesar) ∨ ¬tryassassinate(Marcus, Caesar)
¬ruler(Caesar) ∨ ¬tryassassinate(Marcus, Caesar)
4
¬tryassassinate(Marcus, Caesar)
8
□

Example 2:


Axioms in clause form:

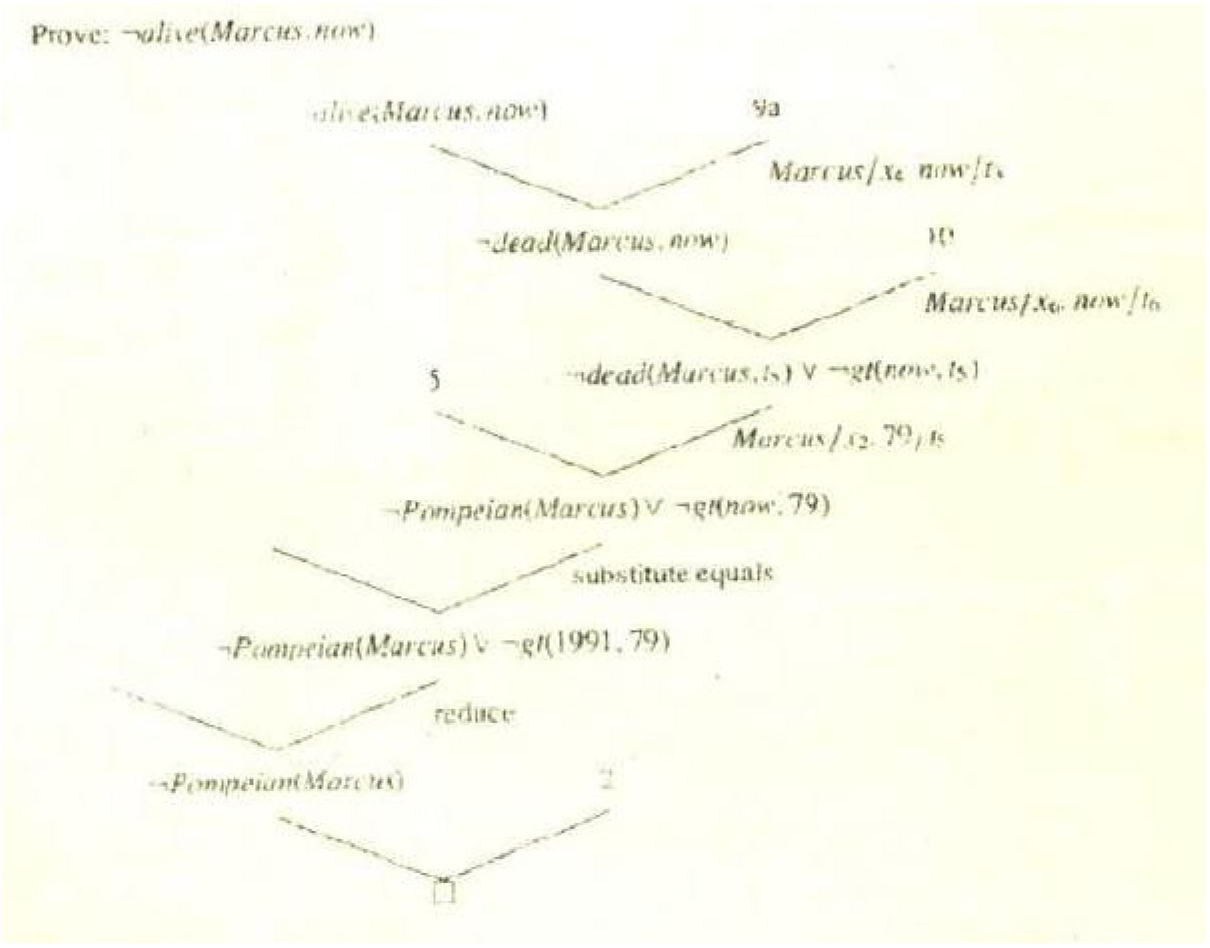
1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\neg man(x_1) \vee mortal(x_1)$
5. $\neg Pompeian(x_2) \vee died(x_2, 79)$
6. $erupted(volcano, 79)$
7. $\neg mortal(x_3) \vee \neg born(x_3, t_1) \vee \neg gt(t_2 - t_1, 150) \vee dead(x_3, t_2)$
8. $now = 1991$

9a. $\neg alive(x_4, t_3) \vee \neg dead(x_4, t_3)$

9b. $dead(x_5, t_4) \vee alive(x_5, t_4)$

10. $\neg died(x_6, t_5) \vee \neg gt(t_6, t_5) \vee dead(x_6, t_6)$

Resolution Proof:



3.13 Natural Deduction:

- We used resolution as an easily implementable proof procedure that relies for its simplicity on a uniform representation of the statements it uses.
- Unfortunately, in uniformity, everything looks the same. Since everything looks the same, there is no easy way to select those statements that are the most likely to be useful in solving a particular problem.
- In converting everything to clause form, we often lose valuable heuristic information that is contained in the original representation of the facts.
- For example, suppose the fact: **All judges who are not crooked are well-educated**, can be represented as:

$$\forall x : judge(x) \vee \neg crooked(x) \rightarrow educated(x)$$

In this form, the statement suggests a way of deducing that someone is educated. But when the same statement is converted to clause form:

$$\neg judge(x) \lor crooked(x) \lor educated(x)$$

It is a way of deducing that someone is not a judge by showing that he is not crooked and not educated. Of course, in a logical sense, it is. But it is almost certainly not the best way, or even a very good way, to go about showing that someone is not a Judge. The heuristic information contained in the original statement has been lost in the transformation.

- Another problem with the use of resolution as the basis of a theorem-proving system is that people do not think in resolution. Thus it is very difficult for a person to interact with a resolution theorem prover, either to give it advice or to be given advice by it. Since proving very hard things is something that computers still do poorly, it is important from a practical standpoint that such interaction be possible.
- **Natural deduction** is describes a melange of techniques used in combination to solve problems that are not tractable by any one method alone.
- One common technique is to arrange knowledge, not by predicates, but rather by the objects involved in the predicates.

Unit- III

Artificial Intelligence: Logic Concepts

(Open Elective –I)

Assignment-Cum-Tutorial Questions

Objective Questions

1. List standard logic symbols.
2. Represent **“It is RAINING”** in propositional logic.
3. Real-world facts written as logical propositions are called _____________.
4. Represent **“Marcus was a man”** in propositional and predicate logic.
5. In ________________ logic decision procedure does not exist. [ ]

(a) Propositional (b) Predicate (c) First order (d) none

6. ____________ and __________ attributes play an important role in process of reasoning.
7. _______________property uses the attributes isa and instance. [ ]

(a) Polymorphism (b) Inheritance (c) Reasoning (d) both b & c

8. The attribute **instance** is _________ predicate. [ ]

(a) Unary (b) Binary (c) Ternary (d) none

9. Resolution produces proofs by __________________. [ ]

(a) Contradiction (b) Reasoning (c) Refutation (d) none

10. Define the term “Resolution”.
11. Moving all quantifiers to the left of the formula without changing their relative order is called ____________________.
12. After resolving the below facts, we get____________________.

winter V summer

¬winter V cold

13. The resultant clause after resolution is called_____________. [ ]

(a) Implicant (b) Resolvent (c) fact (d) instance

14. The substitutions in Unification algorithm are applied from __. [ ]

(a) right to left (b) left to right (c) any order (d) none

15. Resolving clauses that have a single literal first is called _____________ strategy.
16. Resolving either with one of- the clauses that is part of the statement we are trying to refute or with a clause generated by a resolution with such a clause- This is called the ___________ strategy.

II) Descriptive Questions

1. Represent the following sentences in predicate logic:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
S, All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

2. Answer the question: **Was Marcus loyal to Caesar?** using the above facts.

3. Represent the following facts using isa and instance attributes:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
4. Enumerate the steps in the process of resolution.

5. Explain the steps of converting a fact to Clause form.

6. Convert the following fact into clause form:
All Romans who know Marcus either hate Caesar or think that anyone who hates anyone is crazy.
7. Explain the algorithm for Propositional resolution.

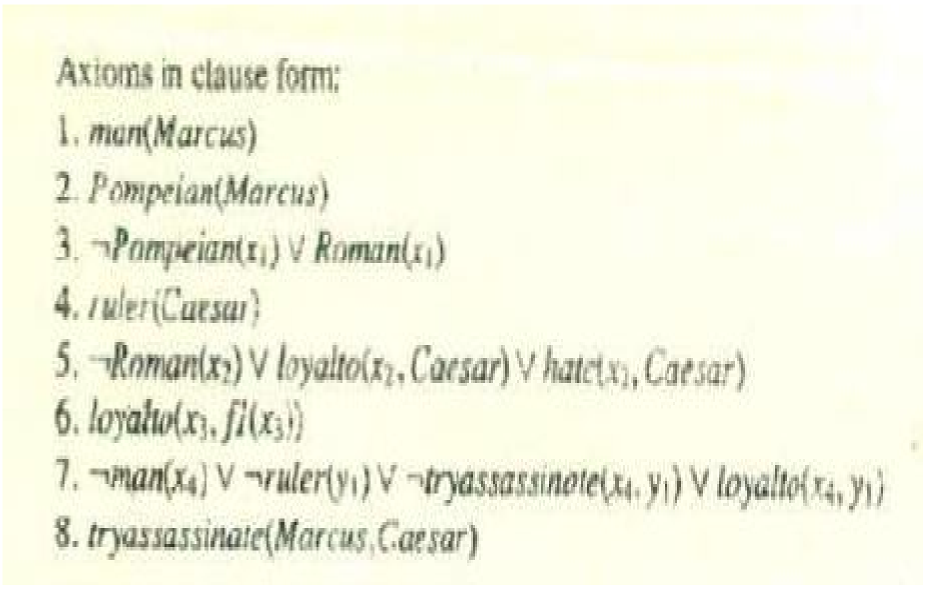
8. Explain Unification Algorithm.

9. Explain the algorithm for resolution in predicate logic.

10. Determine the possible substitutions for unifying the following facts:

hate (x, y)
hate (Marcus, z)

11. Perform resolution in predicate logic:



Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\neg Pompeian(x_1) \vee Roman(x_1)$
4. $ruler(Caesar)$
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$
6. $loyalto(x_3, f1(x_3))$
7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinate(x_4, y_1) \vee loyalto(x_4, y_1)$
8. $tryassassinate(Marcus, Caesar)$

12. Explain about Natural Deduction?