CHAPTER – 5 – USING PREDICATE LOGIC

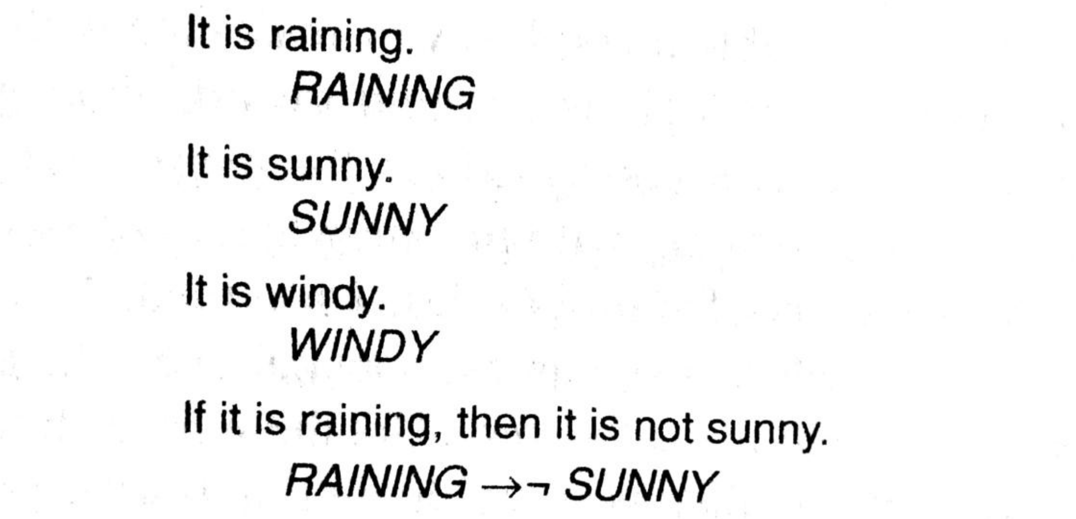
VIVIA MARY JOHN, AP, CSE, CMRIT

TOPICS IN THIS CHAPTER

- REPRESENTING SIMPLE FACTS IN LOGIC
- REPRESENTING INSTANCE AND ISA RELATIONSHIPS
- COMPUTABLE FUNCTIONS AND PREDICATES
- RESOLUTION
- NATURAL DEDUCTION

REPRESENTING SIMPLE FACTS IN LOGIC

- Propositional logic is appealing because it is simple to deal with and a decision procedure for it exists.
- Easily represent real-world facts as logical propositions written as well as well-formed formulas in propositional logic.



It is raining.

RAINING

It is sunny.

SUNNY

It is windy.

WINDY

If it is raining, then it is not sunny.

$RAINING \rightarrow \neg SUNNY$

Fig. 5.1 *Some Simple Facts in Prepositional Logic*

REPRESENTING SIMPLE FACTS IN LOGIC

- Socrates is a man: SOCRATESMAN
- Plato is a man: PLATOMAN
- These are totally separate assertions and we would not be able to draw any conclusions about similarities.
- Better representation:
  - MAN(SOCRATES)
  - MAN(PLATO)
- Structure of representation reflects structure of the knowledge.

REPRESENTING SIMPLE FACTS IN LOGIC

- Consider the sentence: All men are mortal
- Representation: MORTALMAN
- Fails to capture the relationship between individual being a man and individual being mortal
- Move to first order predicate logic as a way of representing knowledge
  - It permits representation of things that cannot be represented in propositional logic.
- If we use logical statements as a way of representing knowledge, then we can reason with that knowledge.

REPRESENTING SIMPLE FACTS IN LOGIC

- Determine the validity of a proposition in propositional logic
  - Find whether it also provides a good way of reasoning with knowledge.
  - Adopt predicate logic as a good medium for representing knowledge.
  - Provides a way of deducing new statements from old ones
- Procedure is to use the rules of inference to generate theorem from the axioms.

EXPLORING USE OF PREDICATE LOGIC TO REPRESENT KNOWLEDGE

1. Marcus was a man = man(Marcus)
   - Captures critical fact. Fails to capture past tense.
2. Marcus was a Pompeian = Pompeian(Marcus)
3. All Pompeians were Romans = for all x: Pompeian(x) → Roman(x)
4. Caesar was a ruler = ruler(Caesar)
   - Ignore the fact that proper names are pften not references to unique individuals.
5. All Romans were either loyal to Caesar or hated him = for all x: Roman(x) → loyalto(x, Caesar) ∨ hate(x,Caesar)

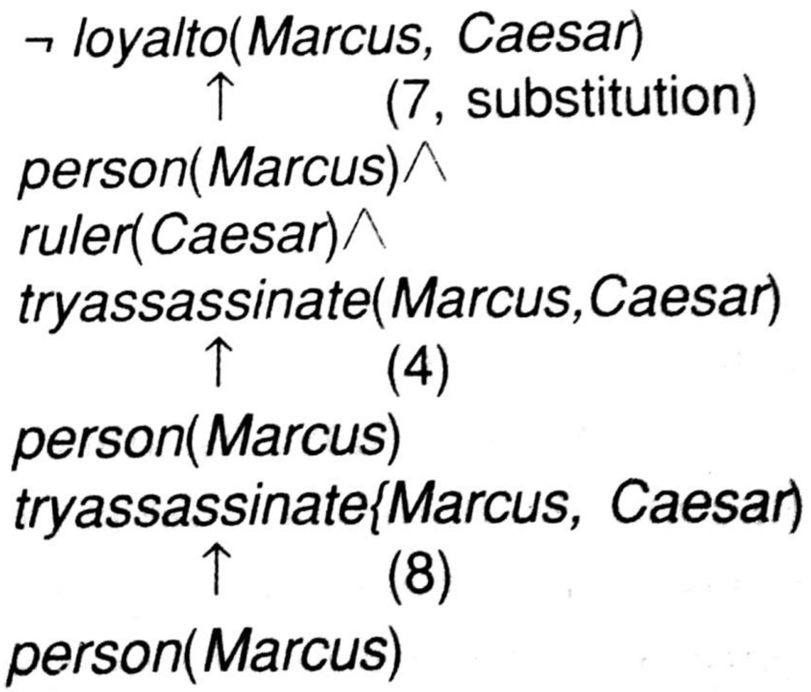
EXPLORING USE OF PREDICATE LOGIC TO REPRESENT KNOWLEDGE

6. Everyone is loyal to someone = for all x: → y: loyalto(x, y)
7. People only try to assassinate rulers they are not loyal to = for all x: for all y: person(x) ∧ ruler(y) ∧ tryassassinate(x, y) → ¬ loyalto(x, y)
8. Marcus tried to assassinate Caesar = tryassassinate(Marcus, Caesar)

Was Marcus loyal to Caesar? Using 7 and 8, we can prove that Marcus was not loyal to Caesar (again ignoring present tense and past tense): ¬ loyalto(Marcus, Caesar)

EXPLORING USE OF PREDICATE LOGIC TO REPRESENT KNOWLEDGE

9. All men are people = for all x: man(x) → person(x)



$$\begin{tikzcd}
\neg loyalto(Marcus, Caesar) \\
person(Marcus) \land ruler(Caesar) \land tryassassinate(Marcus, Caesar) \arrow[u, "{(7, \text{substitution})}"'] \\
person(Marcus)\ tryassassinate(Marcus, Caesar) \arrow[u, "(4)"'] \\
person(Marcus) \arrow[u, "(8)"']
\end{tikzcd}$$

Fig. 5.2 *An Attempt to Prove ¬loyalto(Marcus,Caesar)*

ISSUES OF USING PREDICATE LOGIC TO REPRESENT KNOWLEDGE

- Three issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones
  - Many English sentences are ambiguous
  - There is often a choice of how to represent the knowledge. Simple representations are desirable, but they may preclude certain kinds of reasoning.
  - A set of sentences is unlikely to contain all the information necessary to reason about the topic at hand.
- When asking the computer program: Was Marcus loyal to Caesar?

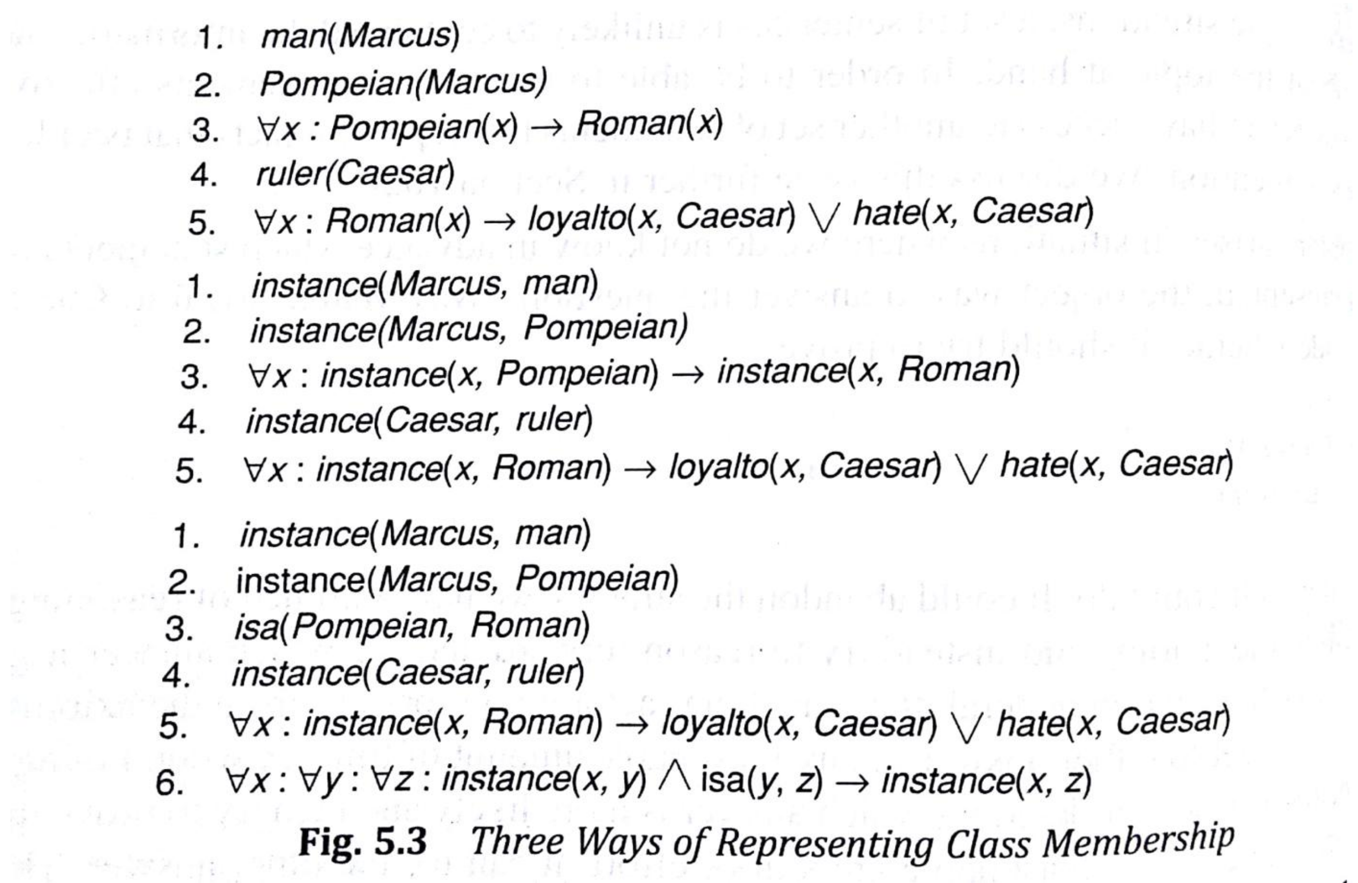
The computer program does not know whether to prove loyalto(Marcus, Caesar) or ¬ loyalto(Marcus, Caesar)

ISSUES OF USING PREDICATE LOGIC TO REPRESENT KNOWLEDGE

- Several methods to avoid these issues
  - Abandon the strategy of backward reasoning.
  - Try forward reasoning to reach a goal
  - Use heuristic search for determining which answer is more likely and work towards that.
  - Try proving both answers and stop when one effort is successful
  - Try proving one answer. Disprove it. Use information gained in one of the process to guide the other.

REPRESENTING INSTANCE AND ISA RELATIONSHIPS

- In Marcus and Caesar problem, these attributes have not been used – the relationships necessary have been captured.



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. $\forall x : \forall y : \forall z : instance(x, y) \wedge \mathrm{isa}(y, z) \rightarrow instance(x, z)$

Fig. 5.3 *Three Ways of Representing Class Membership*

REPRESENTING INSTANCE AND ISA RELATIONSHIPS

- First part – contains representations we already solved.
  - Class membership is represented with unary predicates (Roman), each of which corresponds to a class.
  - Asserting that P(x) is true is equivalent to asserting that x is an instance (element) of P
- Second part – contains representations that use the instance predicate explicitly.
  - Predicate instance is binary.
    - First argument is object
    - Second argument is a class to which the object belongs.
  - In sentence 3, if an object is of subclass Pompeian, it is of class Roman.

REPRESENTING INSTANCE AND ISA RELATIONSHIPS

- Third part – contains representations that use both instance and isa predicates explicitly
  - Use of the isa predicate simplifies the representations of sentence 3, but it requires an additional axiom (Sentence 6)
    - Additional axiom describes how an instance relation and an isa relation can be combined to derive a new instance relation.
    - Additional axiom is general and does not need to be provided separately for additional isa relations.

REPRESENTING INSTANCE AND ISA RELATIONSHIPS

- This example illustrates 2 points:
  - Although class and super-class memberships are important facts that need to be represented
    - Those memberships need not be represented with predicates labeled instance and isa.
    - Instead unary predicates corresponding to the classes are often used.
  - There are different ways of representing a given fact within a particular representational framework.
    - The choice depends partly on which deductions need to be supported most efficiently and partly on taste.
    - Within a knowledge base, consistency of representation is crucial.
- Necessary that all the knowledge to which that rule is intended to apply be in the form that the rule demands.

REPRESENTING INSTANCE AND ISA RELATIONSHIPS

- By permitting the inference of super-class membership, they permit the inference of other properties associated with membership of that super-class.
- Procedure for manipulating isa hierarchy guaranteed that we always found the correct value for any attribute.
- We cannot add these facts to our existing knowledge base the way we could just add new nodes into a semantic net.
- In this framework, every exception to a general rule must be stated twice
  - once in a particular statement
  - once in an exception list that forms part of the general rule.

COMPUTABLE FUNCTIONS AND PREDICATES

- All the simple facts were expressed as combinations of individual predicates.
  - This is fine if the number of facts is not very large
  - If the facts themselves are sufficiently unstructured that there is little alternative.
- Useful to augment our representation by these computable predicates.
- Whatever proof procedure we use, when it comes upon one of these predicates
    - instead of searching for it explicitly in the database or
    - attempting to deduce it by further reasoning
  - invoke a procedure, which we will specify in addition to our regular rules,
  - That will evaluate it and return true or false.

COMPUTABLE FUNCTIONS AND PREDICATES

- Consider the following set of facts
  1. Marcus was a man = man(Marcus)
  2. Marcus was a Pompeian = Pompeian(Marcus)
  3. Marcus was born in 40 A.D. = born(Marcus, 40)
  4. All men are mortal = for all x: man(x) → mortal(x)
  5. All Pompeians died when the volcano erupted in 79 A.D. = erupted(volcano, 79) ∧ for all x: [Pompeian(x) → died(x, 79)]
  6. No mortal lives longer than 150 years =

For all x: for all t_1: for all t_2: mortal(x) ∧ born(x, t_1) ∧ gt($t_2 - t_1$, 150) → dead(x, t_2)

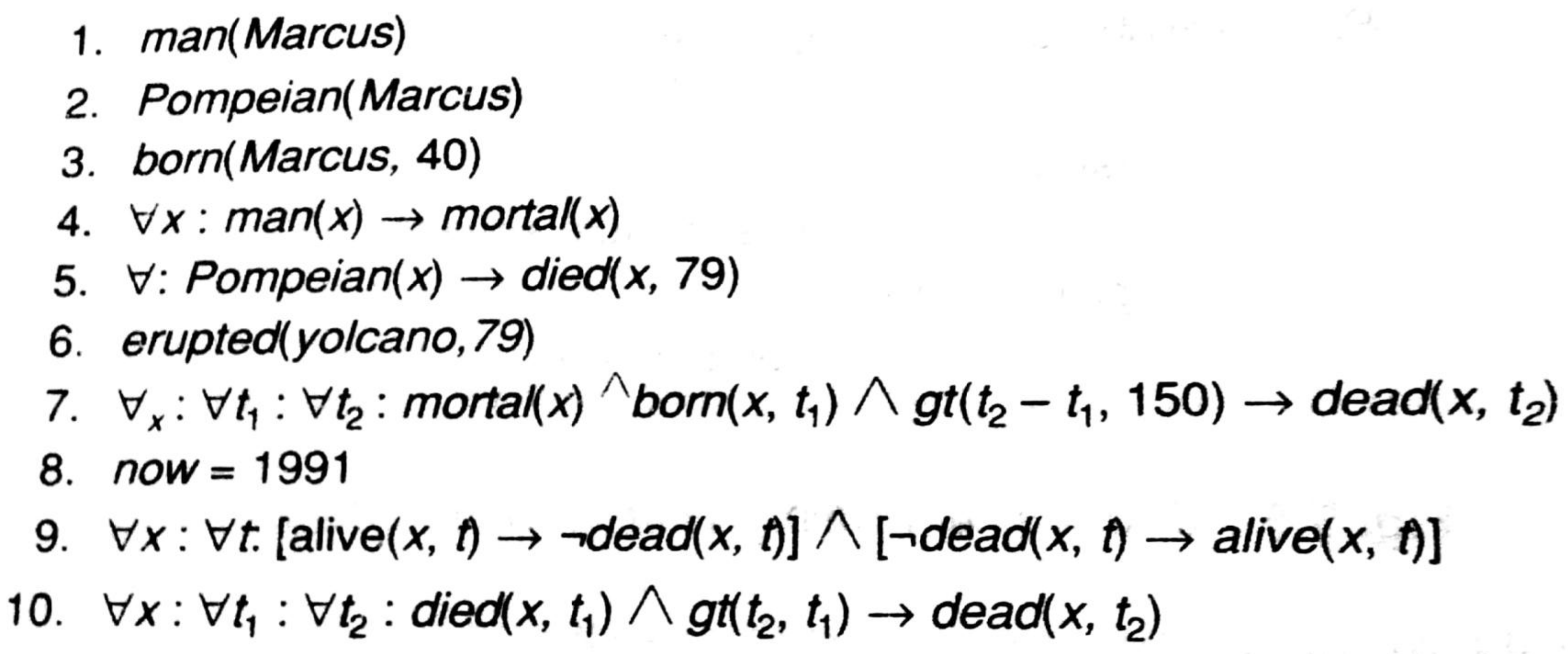
  7. It is now 1991 = now = 1991
  8. Alive means not dead =

for all x: for all t: [alive(x,t) →¬ dead(x,t)] ∧ [¬dead(x,t) → alive(x,t)]

COMPUTABLE FUNCTIONS AND PREDICATES

9. If someone dies, then he is dead at all later times =

For all x: for all t_1: for all t_2: died(x,t_1) $\wedge$ gt(t_2,t_1) $\rightarrow$dead(x,t_2)

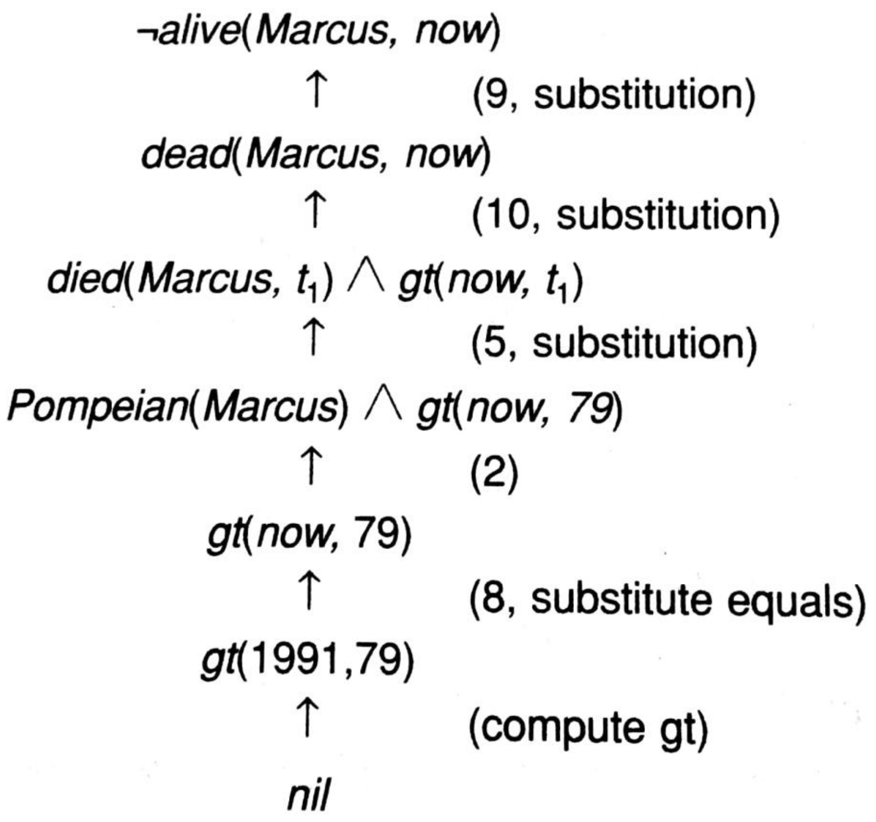


1. *man*(*Marcus*)
2. *Pompeian*(*Marcus*)
3. *born*(*Marcus*, 40)
4. $\forall x : man(x) \rightarrow mortal(x)$
5. $\forall$: *Pompeian*(*x*) $\rightarrow$ *died*(*x*, 79)
6. *erupted*(*yolcano*,*79*)
7. $\forall_x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$
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. $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$

Fig. 5.4 *A Set of Facts about Marcus*

COMPUTABLE FUNCTIONS AND PREDICATES

- Is Marcus alive? Prove ¬alive(Marcus, now)



$$\begin{array}{ll}
\neg alive(Marcus,\ now) & \\
\uparrow & (9,\ \text{substitution}) \\
dead(Marcus,\ now) & \\
\uparrow & (10,\ \text{substitution}) \\
died(Marcus,\ t_1) \wedge gt(now,\ t_1) & \\
\uparrow & (5,\ \text{substitution}) \\
Pompeian(Marcus) \wedge gt(now,\ 79) & \\
\uparrow & (2) \\
gt(now,\ 79) & \\
\uparrow & (8,\ \text{substitute equals}) \\
gt(1991,79) & \\
\uparrow & (\text{compute gt}) \\
nil &
\end{array}$$

Fig. 5.5 *One Way of Proving That Marcus Is Dead*

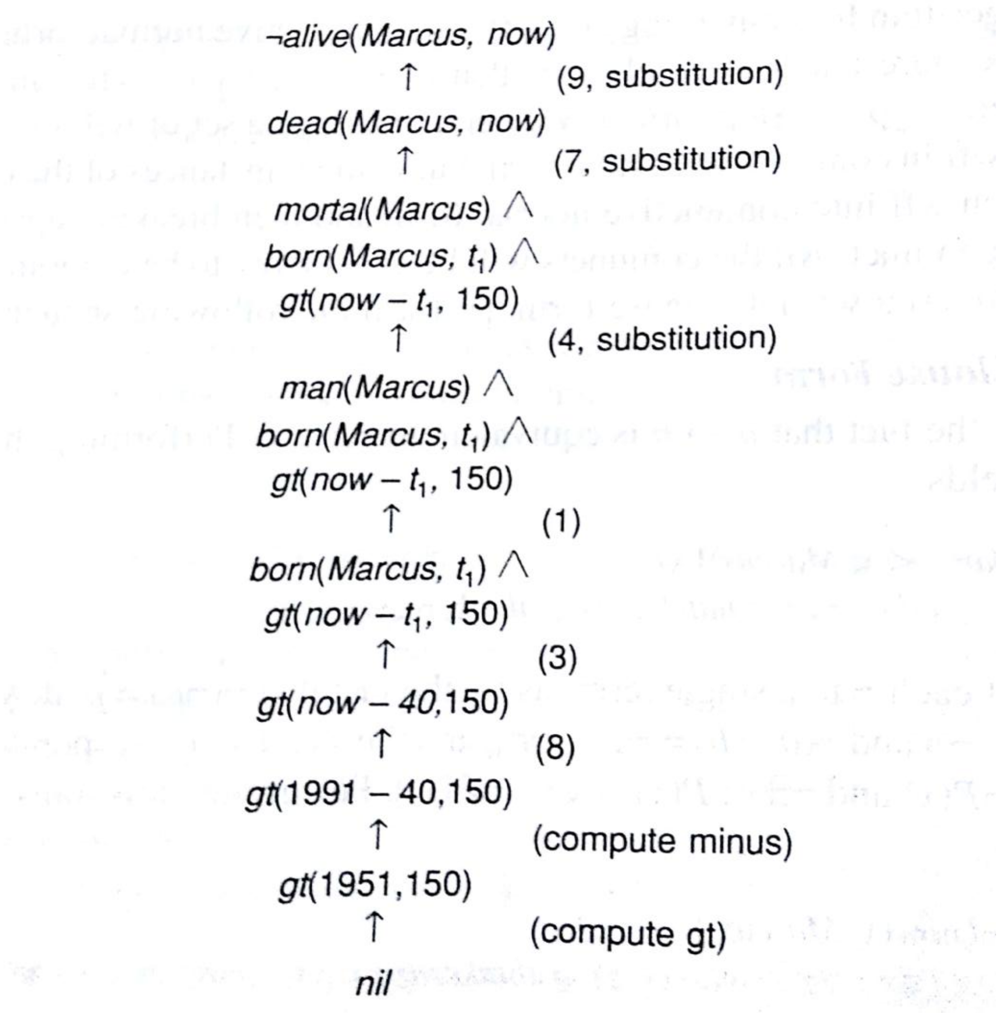
COMPUTABLE FUNCTIONS AND PREDICATES

- Even very simple conclusions can require many steps to prove
  - If we want to do non trivial reasoning, statements allowing us to take bigger steps are necessary.
  - Should represent the facts that people gradually acquire as they become experts.
- A variety of processes – matching, substitution, application – are involved in the production of a proof.
  - Actually building a program to do what people do in producing proofs is not easy.
  - Resolution reduces complexity as it operates on statements that have been converted into a single canonical form.

RESOLUTION

- Proof procedure that carried out in a single operation the variety of processes involved in reasoning with statements in predicate logic.
- Gains it efficiency from the fact that it operates on statements that have been converted to a very convenient standard form
- Produces proofs by refutation
  - To prove a statement, resolution attempts to show that the negation of the statement produces a contradiction with the known statements.
  - Contrasts with the technique that we have been using to generate proofs by chaining backward from the theorem to be proved to the axioms.

RESOLUTION



$$\begin{tikzcd}
\neg alive(Marcus, now) \\
dead(Marcus, now) \arrow[u, "{(9,\ substitution)}"'] \\
mortal(Marcus) \wedge born(Marcus, t_1) \wedge gt(now - t_1, 150) \arrow[u, "{(7,\ substitution)}"'] \\
man(Marcus) \wedge born(Marcus, t_1) \wedge gt(now - t_1, 150) \arrow[u, "{(4,\ substitution)}"'] \\
born(Marcus, t_1) \wedge gt(now - t_1, 150) \arrow[u, "(1)"'] \\
gt(now - 40, 150) \arrow[u, "(3)"'] \\
gt(1991 - 40, 150) \arrow[u, "(8)"'] \\
gt(1951, 150) \arrow[u, "{(compute\ minus)}"'] \\
nil \arrow[u, "{(compute\ gt)}"']
\end{tikzcd}$$

Fig. 5.6 *Another Way of Proving That Marcus is Dead*

Conversion to Clause Form

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

$$\forall x : [Roman(x) \wedge know(x, Marcus)] \rightarrow [hate(x, Caesar) \vee (\forall y : \exists z : hate(y, z) \rightarrow thinkcrazy(x, y))]$$

- The formula would be easier to work with if:
  - It were flatter (less embedding of components)
  - The quantifiers were separated from the rest of the formula
- Conjunctive normal form has these properties:

For e.g., the above sentence would be represented in conjuctive normal form as:

$$\neg Roman(x) \wedge \neg know(x, Marcus) \vee hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, z)$$

ALGORITHM

1. Eliminate →, using the fact that a → b is equivalent to ¬a ∨ b.

$$\forall x : \neg[Roman(x) \wedge know(x, Marcus)] \vee$$
$$[hate(x, Caesar) \vee (\forall y : \neg(\exists z : hate(y, z)) \vee thinkcrazy(x,y))]$$

2. Reduce the scope of each ¬ to a single term.

$$\forall x : [\neg Roman(x) \vee \neg know(x, Marcus)] \vee$$
$$[hate(x, Caesar) \vee (\forall y : \forall z : \neg hate(y, z) \vee thinkcrazy(x, y))]$$

3. Standardize variables so that each quantifier binds a unique variable.

$$\forall x : P(x) \vee \forall x : Q(x)$$

would be converted to

$$\forall x : P(x) \vee \forall y : Q(y)$$

ALGORITHM

4. Move all quantifiers to the left of the formula without changing their relative order.

$$\forall x : \forall y : \forall z : [\neg Roman(x) \vee \neg know(x\ Marcus)] \vee [hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x, y))]$$

5. Eliminate existential quantifiers.

$\exists y : President(y)$

can be transformed into the formula

$President(S1)$

6. Drop the prefix.

$$[\neg Roman(x) \vee \neg know(x, Marcus)] \vee [hate(x, Caesar) \vee (\neg hate(y, z) \vee thinkcrazy\{x, y))]$$

ALGORITHM

7. Convert the matrix into a conjunction of disjuncts.

   $\neg Roman(x) \vee \neg know(x, Marcus) \vee$
   $hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, y)$

8. Create a separate clause corresponding to each conjunct

9. Standardize apart the variables in the set of clauses generated in step 8.

   - During the resolution procedure it is necessary to instantiate a universally quantified variable.

THE BASIS OF RESOLUTION

- The resolution procedure is a simple iterative process.
  - At each step, two clauses – parent clauses – are compared, 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 ∨ Summer
  - ¬Winter ∨ Cold

This means that both the clauses must be true.

THE BASIS OF RESOLUTION

- Let's assume either winter or ¬winter is true.
- If winter is true, then cold must be true to support the second clause.
- If ¬winter is true, then summer must be true to support the first clause.
- From these two clauses, we deduce that wither summer is true or cold is true

i.e., summer ∨ cold

- Resolution operates by taking two clauses that each contain the same literal.
- The literal must occur in positive form in one clause and negative form in another.

THE BASIS OF RESOLUTION

- The resolvent is obtained by combining al of the literals of the two parent clauses except the ones that cancel.
- If the clause that is produced is an empty clause then a contradiction can be found.

For e.g., winter and ¬winter will produce the empty clause.

- If a contradiction exists, it will be found.
- If no contradiction exists, the procedure might never terminate.

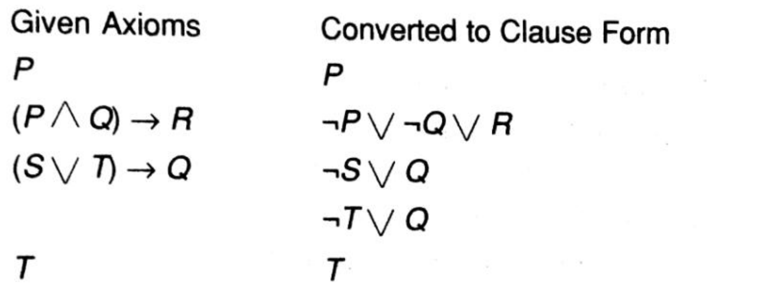
HERBRAND'S THEOREM

- To show that a set of clauses S is unsatisfiable, it is necessary to consider only interpretations over a particular set, called the Herbrand universe of S.
  - One way to go about finding a contradiction is to try systematically the possible substitutions and see if each produces a contradiction.
- A set of clauses S is unsatisfiable if and only if a finite subset of ground instances of S is unsatisfiable.
  - If there exists any computational procedure for proving unsatisfiability, since in a finite amount of time, no procedure will be able to examine a finite set.

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:
  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 them 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: If there are any pairs of literals L and ¬L such that one parent clause contains L and the other contains ¬L, then select one pair and eliminate both L and ¬L from the resolvent.
     c) If the resolvent is the empty clause, then contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

RESOLUTION IN PROPOSITIONAL LOGIC



Given Axioms	Converted to Clause Form
P	P
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$
	$\neg T \vee Q$
T	T

Fig. 5.7 *A Few Facts in Propositional Logic*

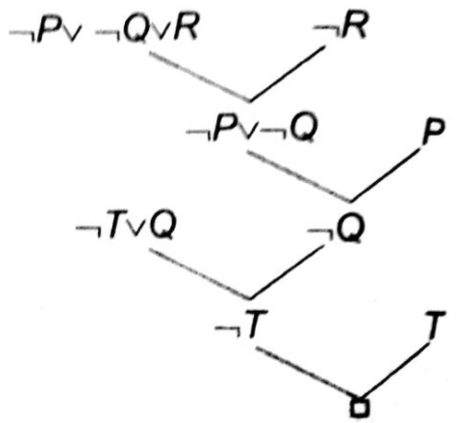


Fig. 5.8 *Resolution in Propositional Logic*

Take a simple example. Suppose the axioms are as given. We want to prove R.

First, we convert the axioms into clause form. Then negate R, producing ¬R, which is already in clause form. Then begin selecting pairs of clauses to resolve together.

One way of viewing the resolution process is that it takes a set of clauses that are all assumed to be true.

A contradiction occurs when a clause becomes so restricted that there is no way it can be true.

In order for proposition 2 to be true is for one of two things to be true: ¬P or ¬Q.

But P os true – proposition 1. We are left with ¬Q to be true. Proposition 4 can be true if either ¬T or Q is true. Already proved ¬Q is true. Then ¬T has to be true. But proposition 5 says T is true. Therefore, we end up with an empty clause – resolvent.

THE UNIFICATION ALGORITHM

- In propositional logic, it is easy to determine that two literals cannot both be true at the same time.
- In order to determine contradictions, need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical.
- Straightforward recursive procedure – unification algorithm.

Basic Idea Of Unification

- To attempt to unify two literals,
  - First check if their initial predicate symbols are same.
    - If so, we can proceed.
    - Otherwise, there is no way they can be unified.

For e.g., the two literals:

- tryassasinate(Marcus, Caesar)
- hate(Marcus, Caesar)

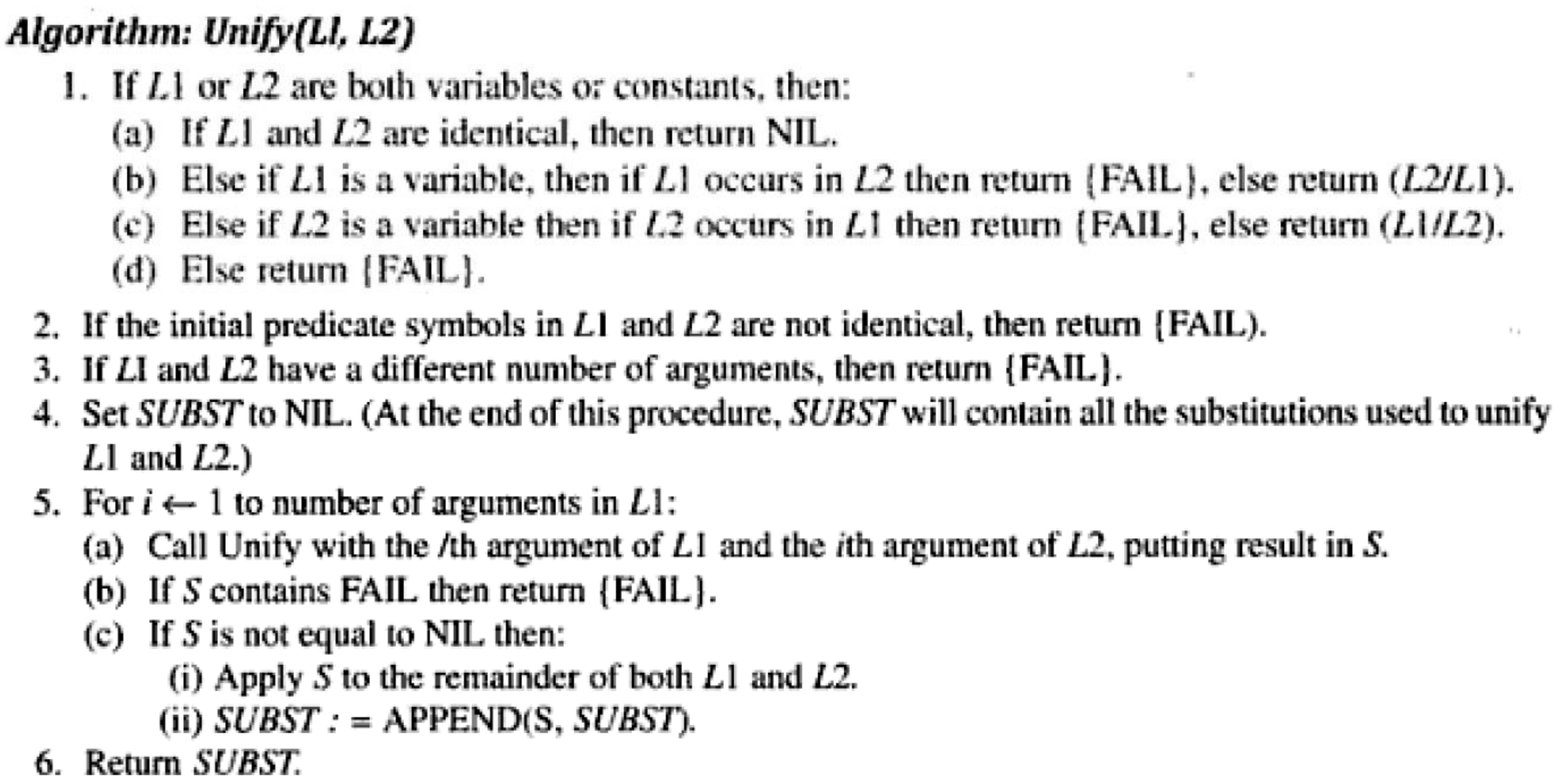
Cannot be unified.

- If the predicate symbols match, then check arguments, one pair at a time.
  - If the first matches, continue with the second, so on.

Basic Idea Of Unification

- To test each argument pair, we can simply call the unification procedure recursively.
- 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
    - The predicate expression must not contain any instances of the variable being matched.
- Complication: we must find a single, consistent substitution for the entire literal, not separate one for each piece of it.

ALGORITHM



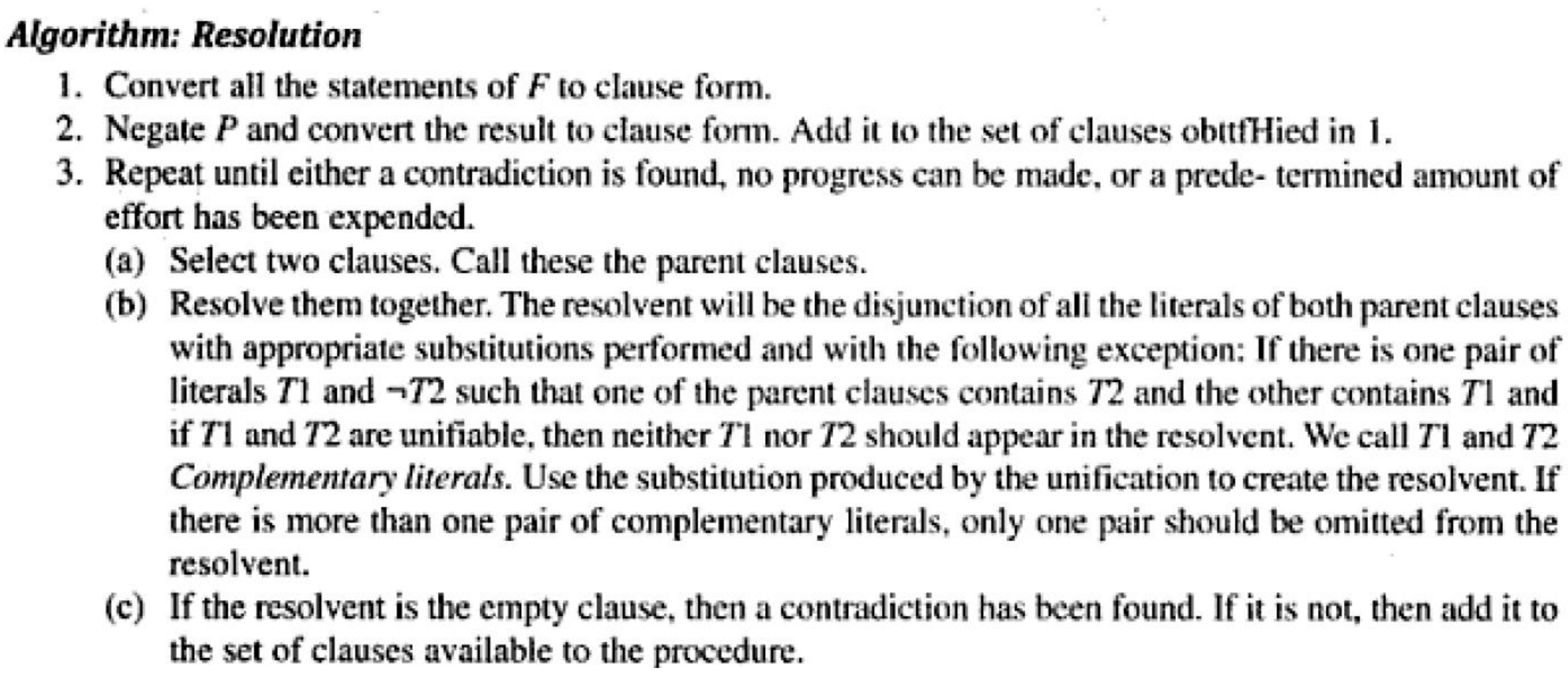
Algorithm: Unify(LI, L2)

```
1. If L1 or L2 are both variables or constants, then:
   (a) If L1 and L2 are identical, then return NIL.
   (b) Else if L1 is a variable, then if L1 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 (L1/L2).
   (d) Else return {FAIL}.
2. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL).
3. If LI 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 L1:
   (a) Call Unify with the /th argument of L1 and the ith argument of L2, putting result in S.
   (b) If S contains FAIL then return {FAIL}.
   (c) If S is not equal to NIL then:
         (i) Apply S to the remainder of both L1 and L2.
        (ii) SUBST : = APPEND(S, SUBST).
6. Return SUBST.
```

RESOLUTION IN PREDICATE LOGIC

- Easy way of determining that two literals are contradictory – they are if one of them can be unified with the negation of the other.
- For e.g., man(x) and ¬man(Spot) are contradictory.
- Corresponds to intuition that man(x) cannot be true for all x if there is known to be some x, say Spot, for which man(x) is false.
- If two instances of the same variable occur, then they must be given identical substitutions.

ALGORITHM



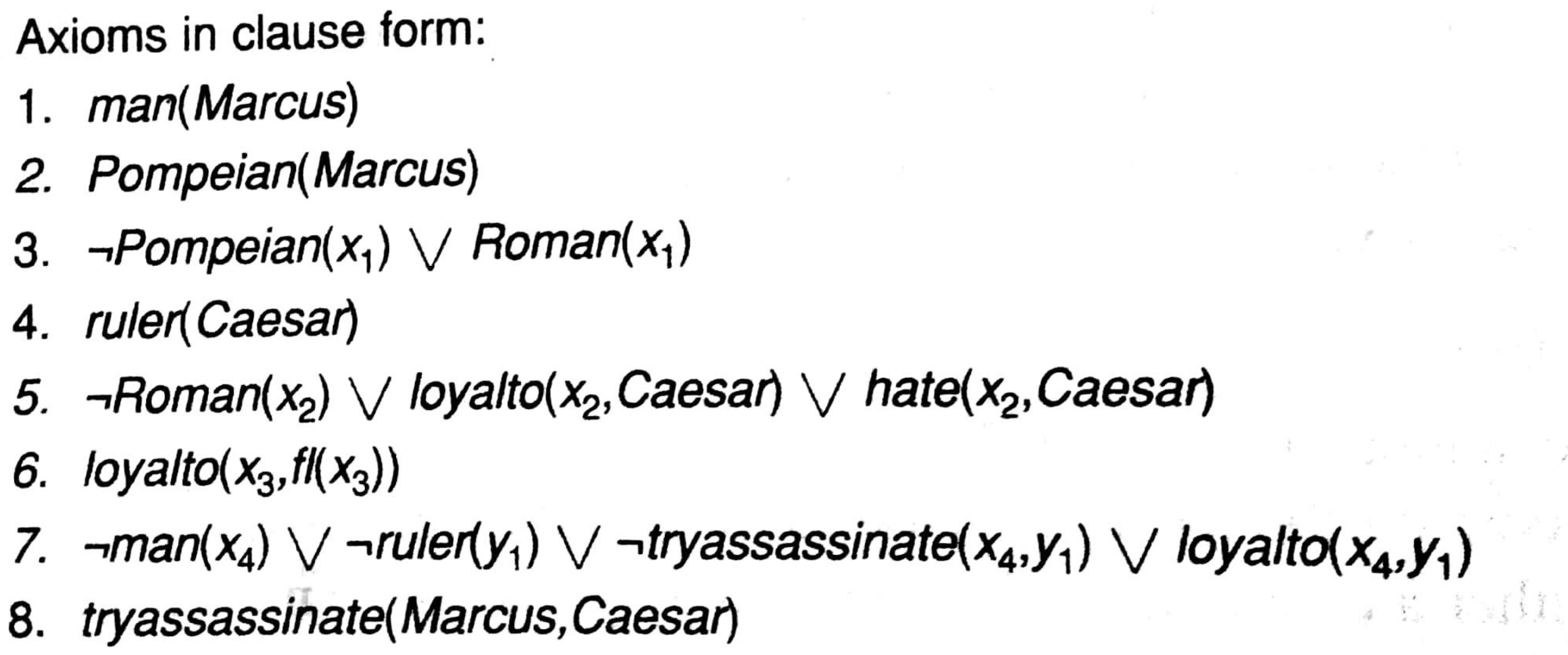
Algorithm: Resolution

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 obttfHied in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a prede- termined 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 *T1* and ¬*T2* such that one of the parent clauses contains *T2* and the other contains *T1* 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 clauses available to the procedure.

RESOLUTION IN PREDICATE LOGIC

- There exists strategies for making the choice that 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.
  - Eliminate certain clauses as soon as they are generated so that they cannot participate in later resolutions.
  - 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.
  - Whenever possible, resolve with clauses that have a single literal – unit preference strategy.

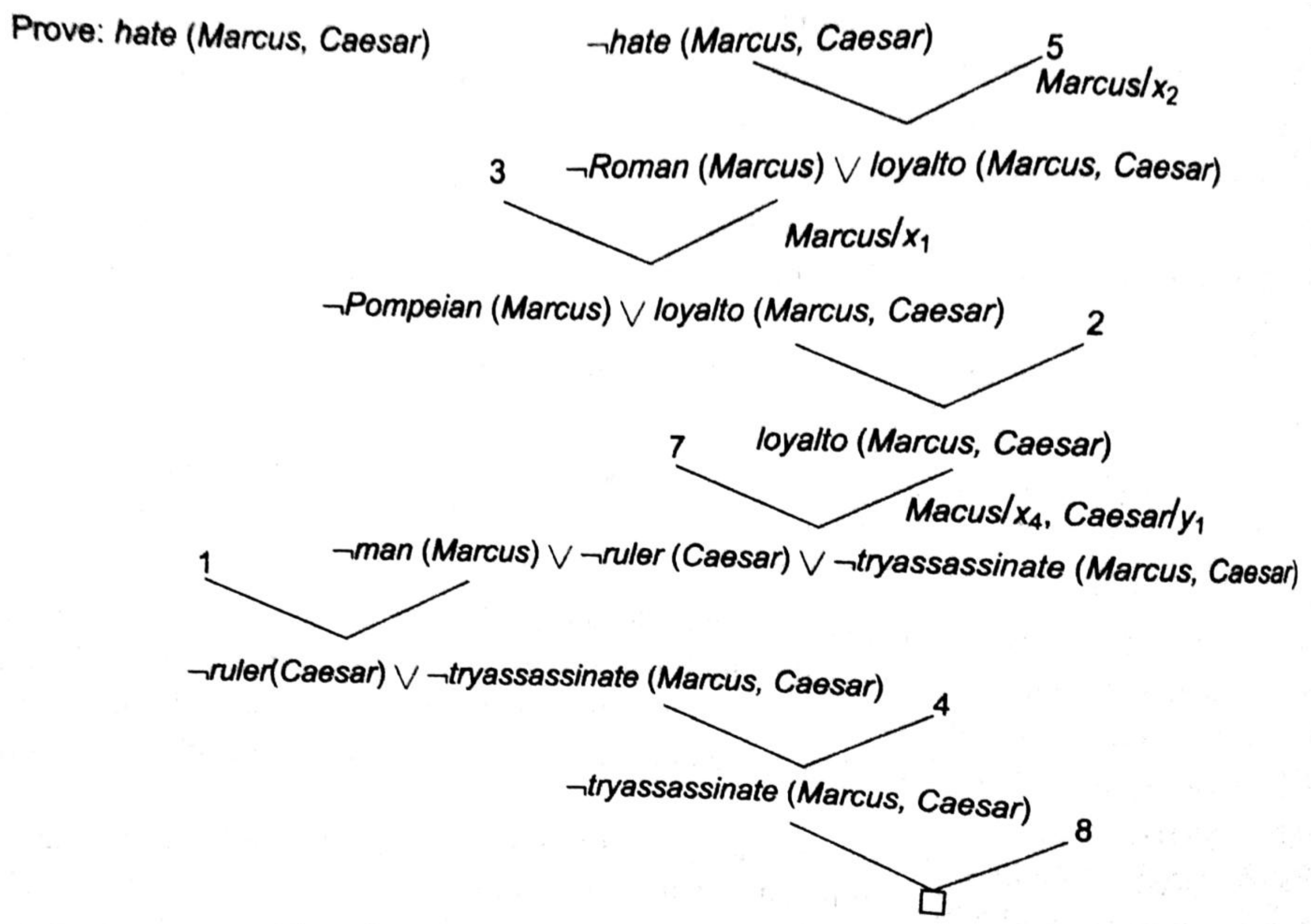
RESOLUTION IN PREDICATE LOGIC



Axioms in clause form:

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

RESOLUTION IN PREDICATE LOGIC



Prove: hate (Marcus, Caesar)
¬hate (Marcus, Caesar)
5
Marcus/x₂
3
¬Roman (Marcus) ∨ loyalto (Marcus, Caesar)
Marcus/x₁
¬Pompeian (Marcus) ∨ loyalto (Marcus, Caesar)
2
7
loyalto (Marcus, Caesar)
Macus/x₄, Caesar/y₁
1
¬man (Marcus) ∨ ¬ruler (Caesar) ∨ ¬tryassassinate (Marcus, Caesar)
¬ruler(Caesar) ∨ ¬tryassassinate (Marcus, Caesar)
4
¬tryassassinate (Marcus, Caesar)
8
□

(b)

Fig. 5.9 *A Resolution Proof*

NEED TO TRY SEVERAL SUBSTITUTIONS

- Resolution provides a very good way of finding a refutation proof
- Does not actually try all the substitutions that Herbrand's theorem suggests.
- Does not always eliminate the necessity of trying more than one substitution.

NEED TO TRY SEVERAL SUBSTITUTIONS

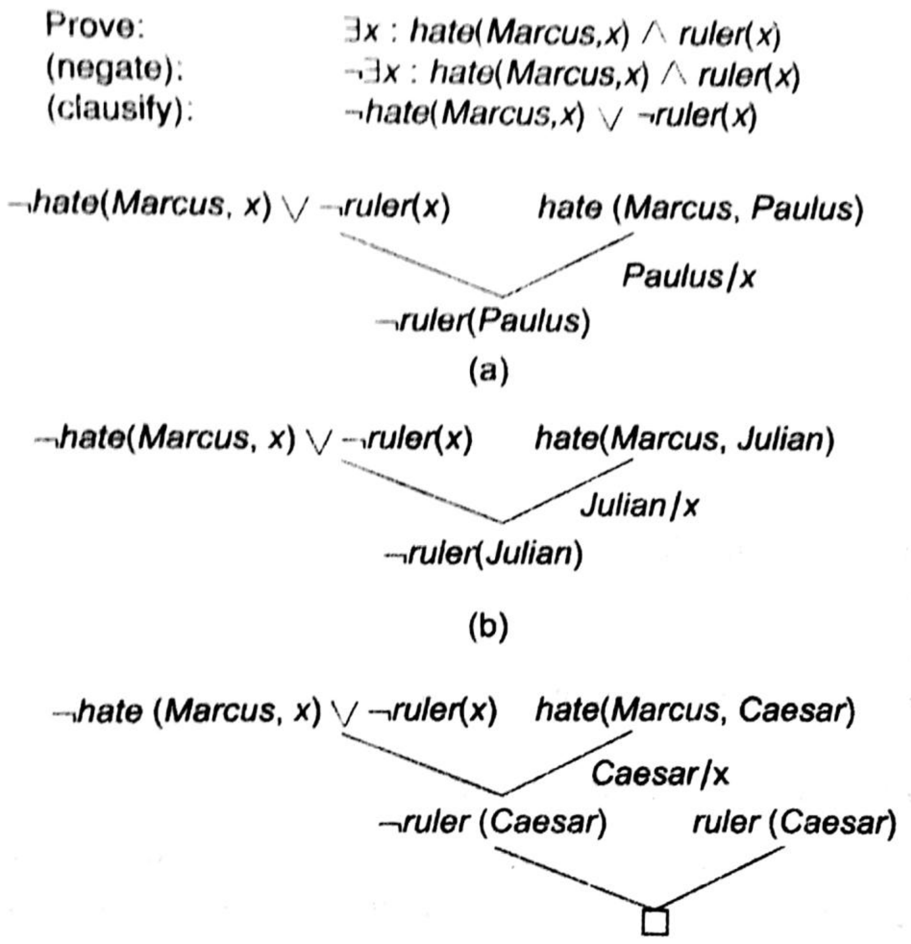
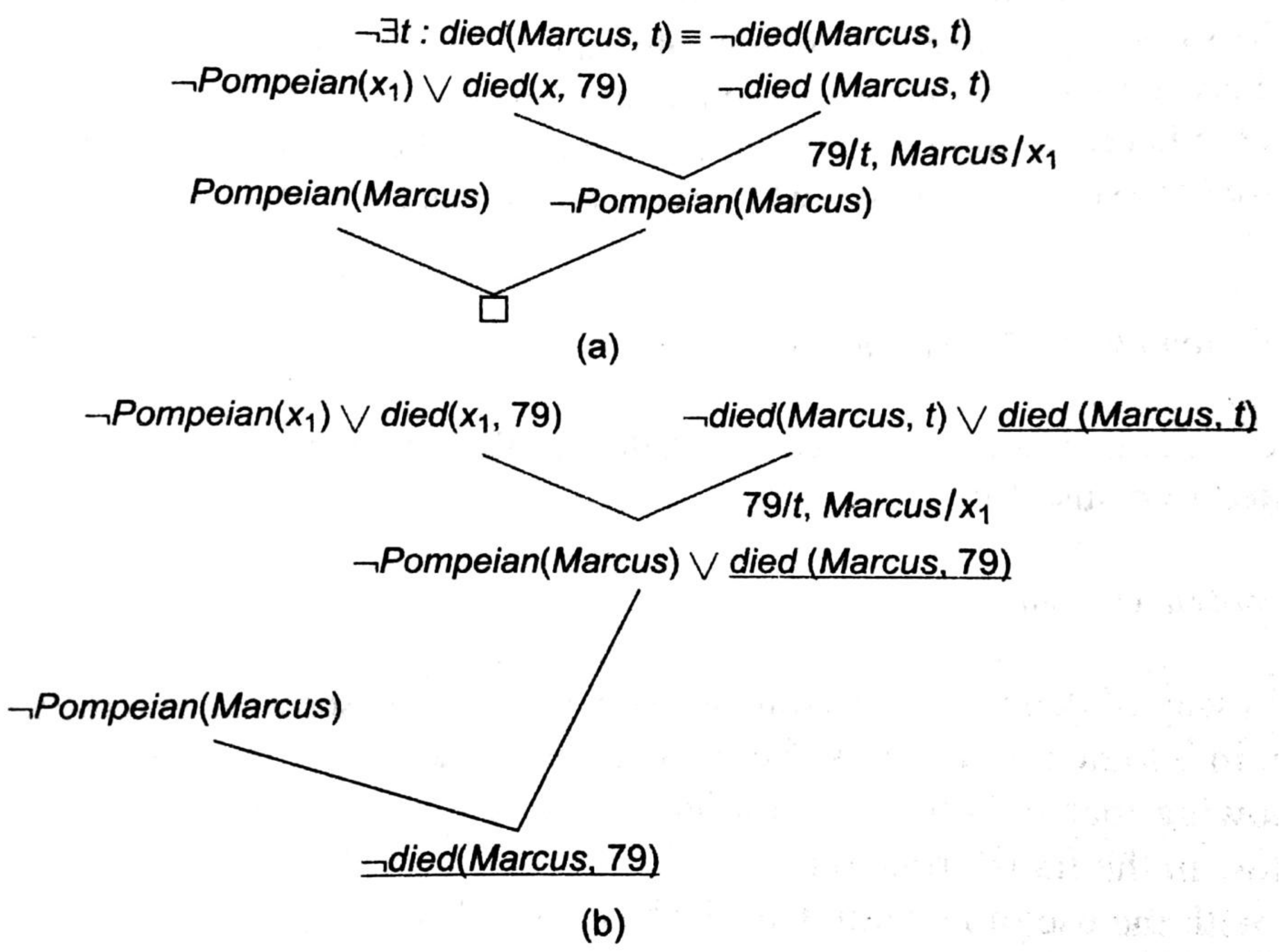

Prove: ∃x : hate(Marcus,x) ∧ ruler(x)
(negate): ¬∃x : hate(Marcus,x) ∧ ruler(x)
(clausify): ¬hate(Marcus,x) ∨ ¬ruler(x)
¬hate(Marcus, x) ∨ ¬ruler(x)
hate (Marcus, Paulus)
Paulus/x
¬ruler(Paulus)
(a)
¬hate(Marcus, x) ∨ ¬ruler(x)
hate(Marcus, Julian)
Julian/x
¬ruler(Julian)
(b)
¬hate (Marcus, x) ∨ ¬ruler(x)
hate(Marcus, Caesar)
Caesar/x
¬ruler (Caesar)
ruler (Caesar)
□
(c)

Fig. 5.13 *Trying Several Substitutions*

QUESTION ANSWERING

- Theorem solving was considered best option for question answering.
- Seems natural – since both deriving theorems from axioms and deriving new facts from old facts – employ deduction.
- The resolution procedure provides an easy way of locating just the statement we need and finds a proof for it.
- The answer to the question can then be derived from the chain of unifications that lead back to the starting clause.

QUESTION ANSWERING



$\neg \exists t : died(Marcus, t) \equiv \neg died(Marcus, t)$
$\neg Pompeian(x_1) \vee died(x, 79)$
$\neg died(Marcus, t)$
$79/t, Marcus/x_1$
$Pompeian(Marcus)$
$\neg Pompeian(Marcus)$
$\Box$
(a)
$\neg Pompeian(x_1) \vee died(x_1, 79)$
$\neg died(Marcus, t) \vee \underline{died(Marcus, t)}$
$79/t, Marcus/x_1$
$\neg Pompeian(Marcus) \vee \underline{died(Marcus, 79)}$
$\neg Pompeian(Marcus)$
$\underline{\neg died(Marcus, 79)}$
(b)

Fig. 5.14 *Answer Extraction Using Resolution*

NATURAL DEDUCTION

- Resolution relies for its simplicity on a uniform representation of the statements.
- Unfortunately, uniformity has it's price – all looks the same.
- No easy way to select those statements that are the most likely to be useful for finding a solution to a problem.
- In converting everything to clause form, we lose valuable information that is contained in the original representation.

NATURAL DEDUCTION

- Another reason – people do not think in resolution.
- Better to look for a way of doing machine theorem proving that corresponds more closely to the processes used in human theorem proving – natural deduction.
- It describes a melange of techniques, used in combination to solve problems that are not tractable by one method alone.
  - One common technique is to arrange knowledge, not by predicates, but by objects involved in the predicates.
  - Another technique is to use a set of rewrite rules that not only describe logical implications but also suggest a way that those implications can be exploited.