Chapter-5

Date __/__/__ Using predicate logic

Saathi

→ Here, we begin exploring one way of representing facts (knowledge) i.e the language of logic.

→ The logical formalism is appealing bcoz it immediately suggests a powerful way of deriving new knowledge from old i.e mathematical deduction.

→ We can conclude that a new statement is true by proving that it follows from the statements that are already known.

→ The truth of an already believed proposition can be extended to include deduction as a way of deriving answers to questions & solutions to problems.

→ The usefulness of some mathematical techniques extends well beyond traditional scope of maths.

☆ Representing simple facts in logic

→ Let's first explore use of propositional logic as a way of representing the world: knowledge that an "AI" system might need.

→ Propositional logic is good, bcoz it is simple to deal with and a decision procedure also exists for it.

→ We can easily represent real-world facts as logical proposition written as "well-formed formulas" in propositional logic, as show in fig.

Date ___/___/_____

Fig. (1)

It is raining.
RAINING

It is sunny.
SUNNY.

It is windy.
WINDY

If it is raining, then it is not sunny.
RAINING → ¬ SUNNY
(NOT)

→ From above fig, we can conclude from the fact that "it is raining", then the fact that "it is not sunny".

→ Suppose we want to represent the obvious fact stated by the classical sentence:
"Socrates is a man."
We can write:- "SOCRATESMAN".
Likewise :- "Plato is a man".
We can write :- "PLATOMAN".

This would be a totally separate assertion, & we would not be able to draw any conclusion about similarities bet" "Socrates" & "Plato".

→ It would be much better to represent these facts as :

MAN (SOCRATES)
MAN (PLATO)

Date __/__/____

Saathi

→ Since now the structure of the repre" reflects the structure of the knowledge itself. Here, we use predicates applied to arguments.

→ We are in even more difficulty if we try to represent the equally classic sentence:-

"All men are mortal."

We could represent this as:

MORTALMAN

→ This fails to capture rela"ship bet" individual being a man & that individual being a mortal. So for this, we really need variables & quantification, if we do not write separate statements about mortality of every man.

→ So, we are forced to move to "predicate logic", since example like above is not suitable for "propositional logic" like fig. (1). In predicate logic, we can represent real-world facts that as "statements" are written in "well-formed formulas" (wff's).

→ If we use logical statements as a way of representing knowledge, then we have good way of reasoning with that knowledge. So, this is a major motivation to use logic.

→ Propositional logic is straightforward, although it is computationally hard.

Date __/__/____

→ Therefore, we are going to adopt "predicate logic". So, before adopting "predicate logic" as a good medium for representing knowledge, we need to ask whether it provides good way of reasoning with knowledge or not. At the first glance, "answer is yes".

→ It provides a way of deducing new statements from old one. "Propositional logic" does not possess decision procedure, but "predicate logic" possess decision procedure.

→ There exists procedures to find a proof of proposed theorem, but these procedures are not guaranteed to halt. In short, "predicate logic" is not decidable, but it is "semi-decidable".

→ ~~There~~ Predicate logic ~~has~~ suffers from negative results, like there can exist no decision procedure for it. So, despite the theoretical undecidability of "predicate logic", it can still serve a useful way of representing & manipulating some kinds of knowledge that an AI system might need.

→ Let's now explore the use of predicate logic as a way of representing knowledge by looking at a specific example. Consider foll. sentences.

(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.
(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 repre^n captures the critical fact of Marcus being a man. It fails to capture some of the information in english sentence, specially the "past-tense". But, this omission is acceptable.

(2) Marcus was a Pompeian.

Pompeian(Marcus)

(3) All Pompeians were Romans.

$\forall x$: Pompeian(x) → Roman(x)

(4) Caesar was a ruler.

ruler(Caesar)

(5) All romans were either loyal to Caesar or hated him.

$\forall x$: Roman(x) → loyal to(x, Caesar) $\vee$
hated(x, Caesar)

Date ___/___/_____

(Saathi)

(6) Everyone is loyal to someone.

$\forall x : \rightarrow y : \text{loyalto}(x, y)$

There can be more appropriate predicate that, "there exists someone to whom everyone is loyal".

$\exists y : \forall x : \text{loyalto}(x, y)$

(7) People only try to assassinate rulers they are not loyal to.

People Ruler

$\forall x : \forall y : \text{person}(x) \wedge \text{ruler}(y) \wedge \text{tryassassinate}(x, y)$

$\rightarrow \neg \text{loyalto}(x, y)$

The above sentence is also "ambiguous".

(8) Marcus tried to assassinate Caesar.

$\text{tryassassinate}(\text{Marcus}, \text{Caesar})$

→ From this brief attempt to convert English sentences into logical statements.

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

"Was Marcus loyal to Caesar?"

→ It seems that using (7) & (8), we should be able to prove that Marcus was not loyal to Caesar. Now, let's try to produce a formal proof, reasoning backward from desired goal.

$\neg \text{loyalto}(\text{Marcus}, \text{Caesar})$

Saathi

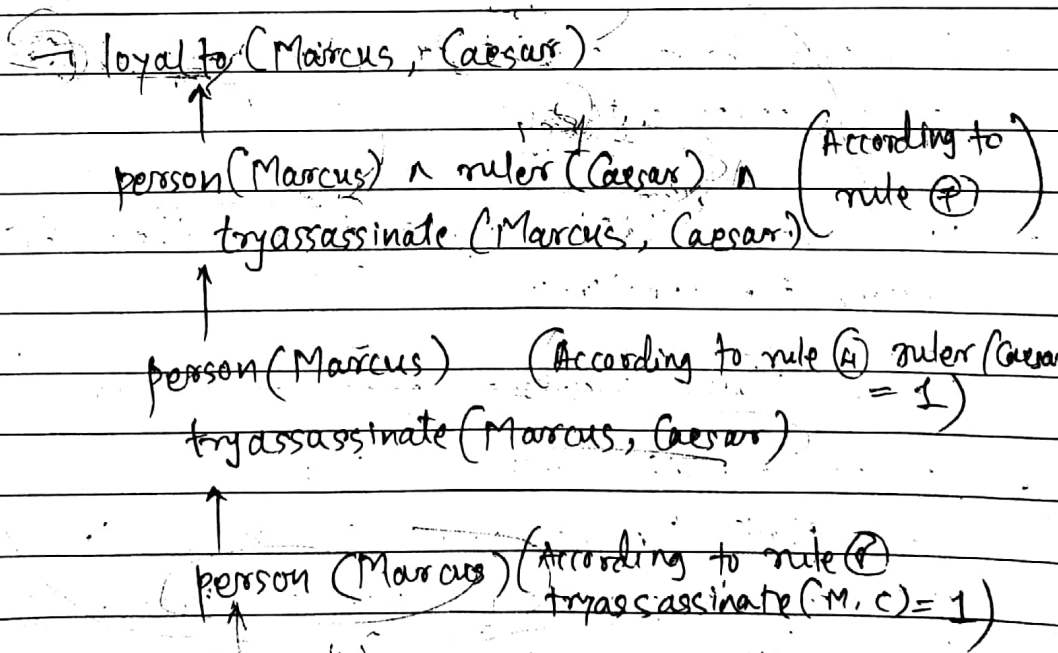
Date __/__/____ ¬loyal to (Marcus, Caesar)

→ In order to prove this goal, we need to use the rules of inference to transform it to another goal that can be transformed, till there are satisfied goals remaining.

→ This process may require the search of an AND-OR graph when there are alternative ways of satisfying individual goals.

→ Fig. shows an attempt to produce a proof of goal by reducing set of unattained goals. This attempt fails, bcoz there is no way to satisfy the goal "person(Marcus)" with some statement that we have available.

→ The problem is that, though we know that "Marcus was a man", we do not have any way to conclude from that "Marcus was a person"



→ We need to add the repre" of another fact to our system, that is...

Date __/__/____

(9) All men are people:

$\forall x$: man(x) $\rightarrow$ person(x)

Now we can satisfy the last goal & produce a proof that Marcus was not loyal to Caesar.

→ From this simple example, we see that three imp. issues must be addressed in process of "converting english sentences into logical statements & then using these statements to deduce new ones:-

(1) Many english sentences are ambiguous like 5, 6 & 7. Choosing correct interpretation may be difficult.

(2) Even in very simple situation, a set of sentences is unlikely to contain all info. necessary to reason about the topic at hand.

→ An additional problem arises in situations, where we don't know in advance which statement to deduce. For ex, "Was Marcus loyal to Caesar?" How to decide that the answer is,

loyalto(Marcus, Caesar)
(OR)
$\neg$loyalto(Marcus, Caesar)

→ It could deny the strategy we have used of reasoning backward from a proposed truth to axioms & see which ans to get.

→ The problem with this approach is that, the branching factor going forward from the axioms is good

Page No.

Date ___/___/___

that it would be probably not yet to answer it in any reasonable amount of time.

☆ Representing instance & is-a relationships

→ We had studied "instance" & "is-a" in a useful form of reasoning, i.e. "property inheritance".

→ But we did not use these attrs like "instance" & "is-a" in the example of "Marcus" & "Caesar". Though we have not used the predicates "instance" & "is-a" explicitly, then also we have captured the relationships they are used to express specially "class membership" & "class inclusion".

→ Fig. shows first five sentences of last section represented in logic in three diff. ways.

(1) The first part of fig. contains the repre^n that we have already discussed. Here, class ~~repre^n~~ membership is represented with unary predicates each of which corresponds to class. Assertion (P(x) = true) is equivalent to assertion that x is an instance of P.

(margin: Unary predicate has only one arg.)

(2) The second part of the fig. contains repre^n that use "instance" predicate explicitly. It's a binary predicate, whose first arg. is an object & 2^nd arg. is a class to which object belongs. But, these repre^n do not use an explicit "is-a" predicate. But instead of "is-a", subclass relationship is also there bet^n "Pompeians" & "Romans" in sentence-3. The implication

(margin: binary predicate has two args. 1st is an obj. & 2nd is class)

Scanned by CamScanner

Date __/__/____

rule states that if an obj is an instance of the subclass "Pompeian", then it's also an instance of superclass "Roman".

(3) The third part contains repre" that use both "instance" & "is-a" predicates. The use of the "is-a" predicate simplifies repre" of sentence (3), but it requires one additional axioms to be provided. (6) The additional axiom describes how an "instance" rela" & an "is-a" rela" can be combined to derive a new "instance" rela"

(1)

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

Without "is-a" & "instance" attr.

Unary

(2)

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

using "instance" attr.

Binary

(3)

1. instance (Marcus, man)
2. instance (Marcus, Pompeian)
3. is-a (Pompeian, Roman)
4. instance (Caesar, ruler)
5. $\forall x$: instance (x, Roman) → loyalto (x, Caesar) ∨ hate (x, Caesar)
6. $\forall x : \forall y : \forall z$: instance (x, y) ∧ is-a (y, z) → instance (x, z)

is-a & instance both

x: any y= Pompeian

x is instance of Roman

Page No.

Date ___/___/_____

★ Computable func"s & predicates

→ Let's consider a simple fact, which was expressed as comb" of individual predicates, such as:-

tryassassinate (Marcus, Caesar)

→ This is fine if no. of facts is not very large or if facts themselves are not sufficiently structured. But, suppose we want to express simple facts, such as foll. "greater than" & "less-than" relationships.

gt(1,0) lt(0,1)
gt(2,1) lt(1,2)
gt(3,2) lt(2,3)
⋮ ⋮

→ Clearly we don't want to write out repre" of these fact individually. They are infinite.

→ But even if we only consider the finite no. of them that can be represented using a single machine word per number, then it would be really inefficient to store explicitly large set of statements.

→ Thus, we have to use "computable predicates". In this, whatever proof procedures we use when it comes upto one of these predicates, then instead of searching for it in DB or deducing it by further reasoning, we

can simply call a procedure, which we will specify in our regular rules, that will evaluate it and return true or false.

→ "Computable func"s" are also useful like "computable predicates". Thus, we might want to be able to evaluate the truth of,

gt(2+3, 1) For this, we first compute the value of plus func" and then send arguments 5 and 1 to "gt."

→ Example-2 "To show how these ideas of computable func"s & predicates can be useful..."

Consider following set of facts involving Marcus:-

(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

$\forall x : man(x) \rightarrow mortal(x)$.

(5) All pompeian died when the volcano erupted in 79 A.D.

erupted (volcano, 79) $\wedge$ $\forall x : [Pompeian(x) \rightarrow died(x, 79)]$

Date ___/___/_____

(6) No mortal lives longer than 150 years

$\forall x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge$
$gt((t_2) - (t_1), 150) \rightarrow dead(x, t_2)$

dead year born year

(7) It is now 1991.

now = 1991

(8) Alive means not dead.

$\forall x : \forall t : [alive(x,t) \rightarrow \neg dead(x,t)] \wedge$
$[\neg dead(x,t) \rightarrow alive(x,t)]$

Computable :

¬alive (Marcus, now)

↑

(8) dead (Marcus, now)

↑

(6) mortal (Marcus) ∧ born (Marcus, t_1) ∧ gt (now − t_1, 150).

↑

Man (Marcus) ∧ bome(Marcus, t_1) ∧ gt (now − t_1, 150)

↑

gt (now − t_1, 150).

↑

(3) gt (now − 40, 150)

↑

gt (1991 − 40, 150)

↑ Compute minus.

gt (1951, 150)

↑ Compute gt

nil.

Date __/__/____

Saathi

X → Conclusion from proofs.

- Even very simple conclusions can require many steps to prove.
- A various processes like, matching, substitution are involved in production of proof. It would be worse if we had more than one term on the right (or) complicated expressions involving "or's" on the left.
- In next section, we introduce a proof procedure called "Resolution" that reduces some of the complexity, bcoz it operates on statements, which are converted to a single canonical form.

S-H ☆ Resolution

→ Resolu^{n} is a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient form.

→ Resolution produces proofs by "refutation." That means, to prove any statement, the reso^{n} attempts to show that the negation of the statement produces a contradiction with the known statements. (2nd method to prove any theorem, i.e reverse will be accepted, then proove it wrong);

Negation of statement produces a contradiction

or ex. to prove
$\sqrt{2}$ is an irrational
., but for that
e accept $\sqrt{2}$ is rational;

→ Here, we are not going to generate proofs by chaining backward from theorem to axioms like previous techniques.

Date ___/___/_____

5.4.

Conversion to clause form

→ Suppose we know that all Romans who know Marcus will either hate Caesar or think that anyone who hates anyone is crazy. Let's represent this in wff.

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

→ To use this formula in a proof requires a complex process. Then, having matched one piece of it, such as thinkcrazy (x, y), it is necessary to do the right thing with the rest of the formula including the pieces in which matched part is embedded & those in which it is not. If the formula were in a simple form, this process would be more easier.

→ It will be easy to work with formula, if
- It were flatter, i.e. there was less embedding of components
- The quantifiers were separated from rest of the formula, so that they did not need to be considered.

→ Conjuctive normal form has both of these properties. For ex: the formula for feelings of Roman who know Marcus would be represented in conjuctive normal form as,

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

Date ___/___/_____

→ Since there exists an algo. for converting any "wff" into conjuctive normal form, there won't be any problem, if we use "resolution" that operates only in wff.

→ In fact, for resoln" to work, we need to go one step further. We need to reduce ea set of wff to a set of clauses."

→ A clause is defined to be a "wff" in conjuctive normal form but with no instances. To convert a wff into clause form, perform foll. sequence of steps.

Algo:- Convert to clause form

1. Eliminate →, using the fact that $a \rightarrow b$ is equivalent to $\neg a \vee b$. Performing this trans" on the "wff" given above yields,

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

2. Reduce '¬' to a single form, using fact that $\neg(\neg P) = P$; De-Morgan's law that $\neg(a \wedge b) = \neg a \vee \neg b$ and $\neg(a \vee b) = \neg a \wedge \neg b$. And standard correspondances bet" quantifiers, $[\neg \forall x : P(x) = \exists x : \neg P(x)$ and $\neg \exists x : P(x) = \forall x : \neg P(x)]$

(quantifier means ∀ ∃)

that means $\neg \forall x : P(x) = \exists x : \neg P(x)$

$\neg \exists x : P(x) = \forall x : \neg P(x)$

∧ = conjuction
∨ = disjunction

Saathi

Date __/__/____

Applying this at step ① yields,

De-Morgan

$\forall x : [\neg Roman(x) \vee \neg know(x, Marcus)] \vee$

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

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

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

would be converted to

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

} This step is in prep. for the next.

∀ and ∃

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

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

$[hate(x, caesar) \vee (\neg hate(y, z) \vee thinkcrazy(x, y))]$

This formula is known as prenex normal form. It consists of a "prefix" of quantifiers followed by a matrix.

∃ elim

5. Eliminate existential quantifiers. A formula contains an existentially quantified variable asserts that there's a value that can be substituted for variables makes the formula true.

Date ___/___/_____

For ex., the formula

$\exists y$: President(y)

can be transformed into the formula

President(S1)

where "S1" is a func^n with no args that somehow produces a value that satisfies "President". For ex. let's say S1 = "India", then there exists the president of india. These generated func^{n}s are called Skolem func^{n}s.

6. Drop the prefix. At this point, all remaining variables are universally quantified, so that the prefix can just be dropped & any proof procedure we can simply assume.

∴ Step (4) appears as :-

$\neg$ Roman(x) $\vee$ $\neg$ know(x, Marcus) $\vee$

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

7. Convert this expression into conjuction of disjunc^{n}s. (ORs of ANDs) Since there's no AND opera^n, so, it will be useful to apply a property. However, it's also frequently necessary to exploit the distributive property [i.e. $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$]:

For ex., the formula

(winter $\wedge$ wearingboots) $\vee$ (summer $\wedge$ wearingsandals)

becomes, after one app^n of the rule,

[winter $\vee$ (summer $\wedge$ wearingsandals)] $\wedge$

[wearingboots $\vee$ (summer $\wedge$ wearingsandals)]

After 2nd app^n, we can still conjuct this by OR's

(winter $\vee$ summer) $\wedge$

(winter $\vee$ wearingsandals) $\wedge$

(wearingboots $\vee$ summer) $\wedge$

(wearingboots $\vee$ wearingsandals)

Date __/__/____

8. Create a separate clause corresponding to each conjuction. For "wff" to be true, all the clauses that are generated from it must be true.

9. Standardize the variables in the set of clauses generated in step (8). That means, "rename" the variables so that no two clauses make reference to same variable. This trans" relies on the fact that,

conjuction

$$(\forall x : P(x) \wedge Q(x)) = \forall x : P(x) \ (\wedge) \ \forall x : Q(x)$$

Thus, since each clause is a separate conjuct & since all variables are universally quantified, there won't be any relationships bet" variables of two clauses, even if they were generated from same "wff".

5.4.2 The Basis of resolution

→ The resolution procedure is a simple iterative process:- at each step, two clauses called as "Parent clauses" will be compared, and obtaining a new clause which is inferred from them.

→ Suppose that there are two clauses in the system:-

winter ∨ summer
¬winter ∨ cold

These both clauses must be true. bcoz if winter is 1, then summer is 0, and in 2nd clause if winter is 1, i.e ¬winter = 0, but if winter=1, then cold = 1.

→ Thus we see that from these two clauses we can deduce

summer ∨ cold

Date ___/___/_____

→ This is the deduction that "resolv" procedure will make. Reso" operates by taking two clauses that each contain same literal, i.e "winter".

→ The literal must occur in positive form in one clause & in negative form in other.

→ The "resolvent" is obtained by combining all the literals but the common one is canceled.

→ If ~~two~~ the clause is produced is empty set, then a "contradiction" is found. For ex., two clauses,

winter

¬ winter

will produce the empty clause.

→ If a contradiction exists, "then nil will be found." Of course, if no contradiction exists, then it is possible that procedure will never terminate.

Saathi

Date __/__/____

5.4.3

* Resolution in propositional logic

→ To make clear that how resolu" works, we first present the resolu" procedure for propositional logic. We the expand it to include predicate logic.

→ In propositional logic, the procedure for producing a proof by resolu" of proposition P with set of axioms F by foll.

X* Algo : Propositional resolu"

1. Convert all propositions of F to clause form.
2. Negate P & convert result to clause form. Add it to 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 is called "resolvent", will be the disjunction (V-OR) of all literals of parent clauses.
   (c) If the resolvent is empty clause, then contradiction is found. If it's not, then add it to the set of clauses available to the procedure.

Suppose we are given the axioms in fig. ① & are asked to prove R.

→ Let's look at an example. First we convert axioms to clause form as shown in second column of fig.

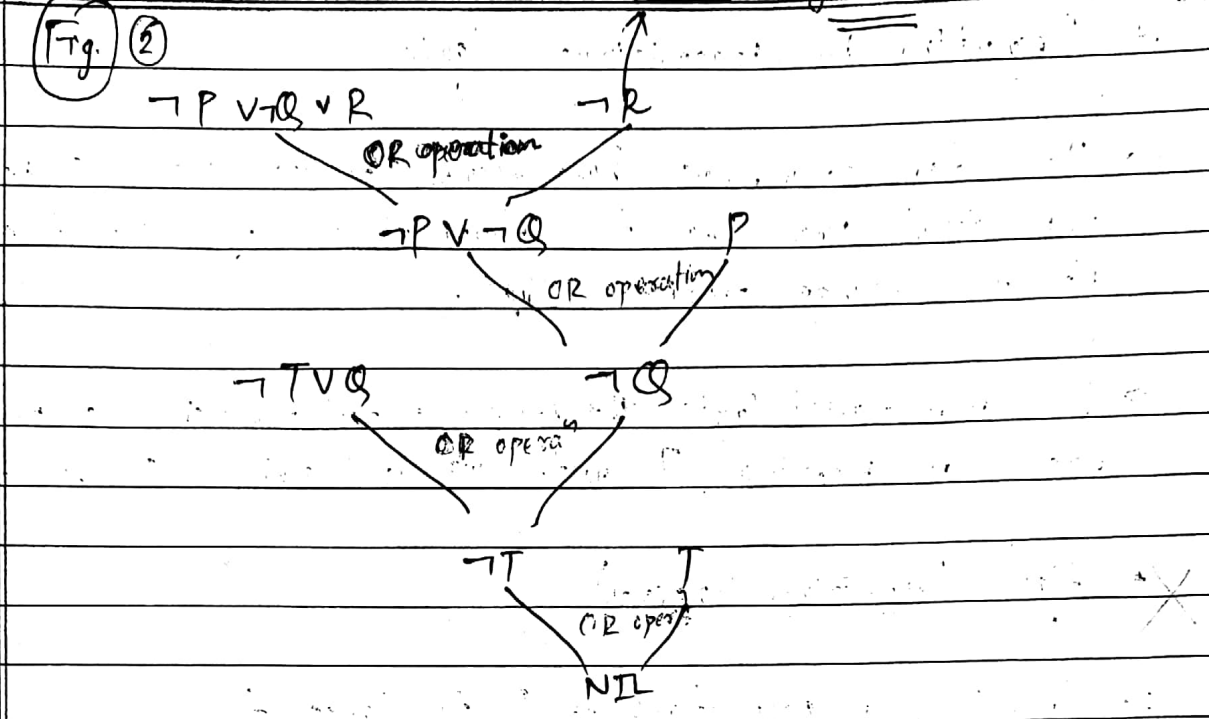
Given Axiom	Converted to clause form
P	P
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$ $(\because a \rightarrow b = \neg a \vee b$
$(S \vee T) \rightarrow Q$	~~$\neg S \wedge \neg T \vee Q$~~ $\neg(S \vee T) \vee Q \Rightarrow \neg S \vee Q$ and $\neg T \vee Q$
T	T

Date __/__/____

Saathi

To be proved, since R is given true

Fig. ②



→ First we negate R by producing ¬R.

→ Then, we begin selecting pairs of clauses to resolve together.

→ Although any pair of clauses can be resolved, but only pairs containing complementary literals (like P and ¬P) will lead to the goal of producing empty set clause.

→ As shown in fig. ② we start by resolving with clause ¬R, since that is one of the clause that must be involved in contradiction.

→ Here, reso^{n} process is that, it takes a set of clauses that are assumed to be true & based on info. provided by others, it generates new clauses.

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

	Axiom	Converted to clause form
①	P	P
②	$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$
③	$(S \vee T) \rightarrow Q$	$\neg S \vee Q$
④		$\neg T \vee Q$
⑤	T	T

Page No.

Date ___/___/___

→ A contra^n occurs when a clause becomes so restricted that there's no way it can be true. This is indicated by generation of "empty" clause.

→ Let's look again at that tree, to see how it works.

(1) In order for propo^n ② to be true, one from $\neg P$, $\neg Q$ or R must be true.

(2) But we assume that $\neg R$ is true.

(3) Now, propo^n ② is true if one from $\neg P$ or $\neg Q$ is true.

(4) But, propo^n ① says that P is true, which mean $\neg P$ can't be true, which leaves only one way for propo^n ② to be true, that is "$\neg Q$" to be true.

(5) Propo^n ④ can be true if either $\neg T$ or Q is true.

(6) But; since we know that "$\neg Q$" must be true, the only way for propo^n ④ to be true is $\neg T$ is true.

(7) But, propo^n ⑤ says that T is true. Thus, there is no way for all clauses to be true in a single interpretation.

5.4.4 ⇒ The unification algo

→ In propositional logic, it is easy to determine that two literals can't both be true at the same time. For ex., for L and $\neg L$ in predicate logic, this matching process is more complicated, since arg.s must be considered of predicate.

→ For ex., man(John) & $\neg$man(John) is a "contradiction", but man(John) & $\neg$man(Spot) is not, since arg.s are different.

→ The main idea for "unification algo." is to determine contradiction. We need a matching procedure that compares two literals & discovers whether there exists a set of

Page No.

Date ___/___/_____

substitution that makes them identical. This is known as a recursive procedure called the "unification algo".

→ The basic idea of unifica" is very simple. To attempt to unify two literals, we first check if their predicate symbols are same. If so, we can proceed. Otherwise, there's no way they can be unified.

→ For ex., the two literals,

tryassassinate (Marcus, Caesar)

hate (Marcus, Caesar)

→ These two can't be unified. If the predicate symbols match, then we must check the arg.s, one pair at a time. If first matches, then we can continue with second and so on:

→ To test each arg. pair, we can simply call the unification procedure recursively. For each arg. pair, we can simply call the unifica" procedure recursively.

* Algo: Unify (L1, L2)

1. If L1 or L2 are both variables or constants, then:-
   (a) If L1 & 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}.

Date __/__/____

2. If initial predicate symbols in L1 & L2 are not identical, then return {FAIL}.
3. If L1 & L2 have a different no. of args, then return {FAIL}.
4. Set SUBST to NIL. (SUBST at end of procedure will contain all substitutions used to identify L1 & L2).
5. For i=1 to no. of args in L1 :-
   (a) Call Unify with the i^{th} arg. of L1 & i^{th} arg. of L2, putting result in S.
   (b) If S = FAIL then return FAIL.
   (c) If S is not equal to NIL; then :-
      (i) Apply S to remainder of both L1 & L2.
      (ii) SUBST = APPEND (S, SUBST).
6. Return SUBST.

* Example of above algo

(1) Suppose we want to unify the expression

P(x, x)

P(y, z)

(2) The two instances of P matches. Next we compare x and y, and decide that to substitute y for x, they could match. We can write it y/x. (See algo. point (1.b))
y for x

(3) If we simply continue and match x and z, we produce the substitution z/x. But we can't substitute both y & z for x, since after that we have not produced a consistent substitution.

(4) After finding the first substitution y/x is to make that substitution throughout the literals by giving,

P(y, y) ← Both x substituted by y.

P(y, z)

Page No.

(5) The obj. of the unifica" procedure is to discover at least one substitution that causes two literals to match. For ex., the literals.

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

This could be unified with any of the foll. substitution:-

(Marcus/x, z/y)
(Marcus/x, y/z)

1.5 ☆ Resolution in predicate logic Q-11

→ We have an easy way of determining that two literals are contradictory, if one of them can be unified with the negation of the other.

→ For ex, man(x) and ¬man(Spot) are contradictory, if x = spot; then man(x) is false. Thus, in order to use resolution for expressions in predicate logic, we use the "unification algo." to locate pairs of literals that cancel out.

→ Suppose we want to resolve two clauses :-
(1) man(Marcus)
(2) ¬man(x_1) ∨ mortal(x_1)

→ The literal man(Marcus) can be unified with literal man(x_1) with the substitution Marcus/x_1, since if x_1 = Marcus, then ¬man(Marcus) = false.

Date __/__/____

→ Clause-2 says that for given x_4, either $\neg man(x_4)$ or $mortal(x_4)$. If $man(Marcus) = true$ in clause-1, then $\neg man(Marcus)$ is false in clause-2. So, we can conclude that $mortal(Marcus)$ must be true for clause-2 to be true.

→ It's not necessary that $mortal(x_4)$ be true for all x_4, since for some values of x_4, $\neg man(x_1)$ might be true, so, $mortal(x_4)$ is irrelevant to the truth of complete clause.

→ So, the resolvent generated by clause-1 & clause-2 must be $mortal(Marcus)$. So, resolution process can then proceed to discover whether $mortal(Marcus)$ leads to a contradiction with other available clauses.

* Algo:- Resolution

1. Convert all the statements of F to clause form.
2. Negate P & convert the result to clause form.
3. Repeat until either a contradiction is found, or no progress is can be made, or a pre-determined amount of effort has been expanded.
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resolvent will be the disjunction of all literals of both parent clauses with appropriate substitutions performed.
   (c) If resolvent is the empty clause, then a contradiction has been found.

Saathi

Date __/__/____

Example

* Let's now return to our discussion of Marcus & show how resolution can be used to prove new things about it. To use them in resolution proofs, we must convert them to clause form as described here. The fig. shows that conversion.

Fig. Axioms in clause form:-

1. man(Marcus)
2. Pompeian(Marcus)
3. ¬Pompeian(x_4) ∨ Roman(x_4)
4. ruler(Caesar)
5. ¬Roman(x_2) ∨ loyalto(x_2, Caesar) ∨ hate(x_2, Caesar)
6. loyalto(x_3, f(x_3))
7. ¬man(x_4) ∨ ¬ruler(y_1) ∨ ¬tryassassinate(x_4, y_1) ∨ loyalto(x_4, y_1)
8. tryassassinate(Marcus, Caesar)

→ Next, fig. shows resolution proof of the statement

Prove:- hate(Marcus, Caesar)

Proof:-

Correct

¬hate(Marcus, Caesar) — (5) OR — Marcus/x_2

(3) ¬Roman(Marcus) ∨ loyalto(Marcus, Caesar) — OR — Marcus/x_4

¬Pompeian(Marcus) ∨ loyalto(Marcus, Caesar) — OR — (2)

(4) loyalto(Marcus, Caesar) — Marcus/x_4, Caesar/y_1

(1) ¬man(Marcus) ∨ ¬ruler(Caesar) ∨ ¬tryassassinate(M, C) — OR

¬ruler(Caesar) ∨ ¬tryassassinate(M, C) — (4) OR

¬tryassassinate(M, C) — (8)

NIL

Page No.

Saathi

Date __/__/____

→ Suppose our actual goal in proving the assertion,

hate(Marcus, Caesar)

was to answer the question: "Did Marcus hate Caesar?"

→ In that case, we attempted to prove the statement,

¬hate(Marcus, Caesar).

→ To do so, we would have added

hate(Marcus, Caesar)

→ If we have attempted to prove ¬hate, then we have to consider "hate", but there are no clause that contains "¬hate".

→ Since the resolⁿ process can only generate new clauses that are composed of combⁿ of literals from existing clauses, we know that hate(Marcus, Caesar) will not produce a contradiction with the known statements.

→ This is an example of a situation in which the resolⁿ procedure does not detect any contradiction.

Saathi

Oct-2012 question

Date __/__/____ Topic-2 Que-2

1. $\forall x$: likes (Rama, x) vegfood(x) → likes(Rama, x)

2. food (ORANGES)

3. food (Mutton)

4. $(\forall x)$: $(\exists y$: eats (y, x) ∧ ¬killed by (y, x)) → food(x)

   → (x ∧ y) = ¬x ∨ ¬y

5. eats (likex, peanuts)

6. alive (likex)

7. $\forall x$: eats (likex, x) → eats (lovex, x)

8. $\forall x : \forall y$: alive (x) → ¬killed (x, y)

* Now let's apply all these sentences into clause form.

1. ¬ vegfood(x) ∨ likes(Rama, x)

2. food (ORANGES)

3. food (Mutton)

4. ¬ eats(y, x) ∨ killed by (y, x) ∨ food(x)

5. eats (likex, peanuts)

6. alive (likex)

7. ¬ eats (likex, x) ∨ eats (lovex, x)

8. ¬ alive(x) ∨ ¬ killed (x, y)

Date __/__/____

Saathi

Que # Prove Rama likes peanuts using Resolution.

$\neg$ likes (Rama, peanuts)

(1) ... peanuts/x

$\neg$ food (peanuts)

(4) ... peanuts/x

$\neg$ eats (Rama, peanuts) V killed by (Rama, peanuts)

(8) ... peanuts/y

$\neg$ eats (y, peanuts) V $\neg$ alive (y)

(6) ... Likes/y

$\neg$ eats (Likes, peanuts)

(5)

NIL (Contradiction)

Hence proved that Rama likes peanuts

Rama likes (Rama, peanuts)

food (peanuts)

eats (y, peanuts) $\wedge$ $\neg$ killedby (y, peanuts)

likes/y

eats (Likes, P) $\wedge$ $\neg$ killedby (Likes, P)

Page No.

Predicate logic question - solution of resolution & backward chaining (Saathi)

Date __/__/____

(2) (10. B)

- Conversion to predicate logic

1. $\forall x$: food (x) → likes (Ram, x)
2. food (Orange)
3. food (Mutton)
4. $\forall x$: $\exists y$: ¬ eats (y, x) ∧ ¬ killed by (y, x) → food (x)
5. eats (Like x, peanuts) ∧ alive (Like x)
6. ~~$\forall x$: eats (Like x, x) →~~
6. $\forall x$: eats (Like x, x) → eats (Love x, x)

- Prove that Rama likes peanust using backward chaining.

likes (Ram, Peanuts)

↑

food (peanuts)

↑

eats (y, peanuts) ∧ ¬ killed by (y, peanuts)

↑

eats (Like x, peanuts) ∧ alive (Like x)

↑

NIL (Proved)

Date __/__/____

Saathi

- **Conversion to clause form**

1. $\neg food(x) \vee likes(Ram, x)$
2. $food(Orange)$
3. $food(Mutton)$
4. $\neg eats(y, x) \vee killed\ by(y, x) \vee food(x)$
5. $eats(Likex, peanuts) \wedge alive(Likex)$
6. $\neg eats(Likex, x) \vee eats(Lovex, x)$

- **Prove Rama likes peanuts using resolution**

$\neg likes(Ram, peanuts)$

(1) peanuts / x

$\neg food(peanuts)$

(4) peanuts / x

$\neg eats(y, peanuts) \vee killed\ by(y, peanuts)$

(5) Likex / y

$eats(Likex, peanuts) \vee \neg eats(y, peanuts)$

$\wedge\ alive(Likex) \vee \neg alive(z, Likex, peanuts)$

NIL

NIL

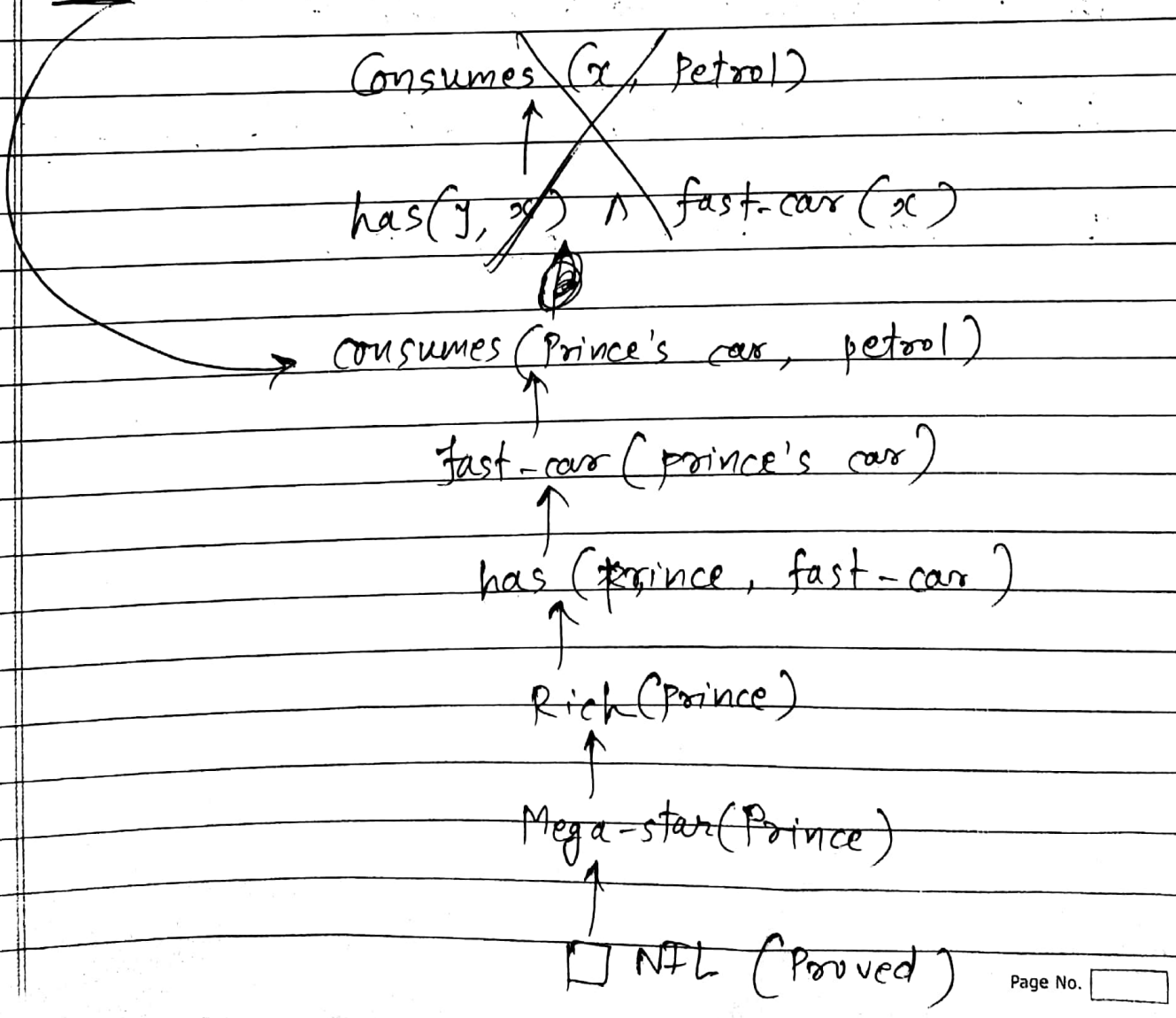
$= NIL \square$

Page No.

- Conversion to predicate logic

1. mega-star (Prince)
2. $\forall x$: Mega-star $(x) \rightarrow$ Rich (x)
3. $\forall x$: Rich $(x) \rightarrow$ $\forall y$: has $(y, x) \wedge$ has $(x,$ fast-car$)$
4. $\forall x$: fast-car $(x) \rightarrow$ consumes $(x,$ petrol$)$

- Translate these sentences
- Prove that Prince's car consumes a lot of petrol.



Page No.

Consider following facts.

Use resolution and prove: If John is a child, then John carves some pumpkin.

1. Every child loves Santa.

$\forall x : child(x) \rightarrow loves(x, Santa)$

clause form : $\neg child(x) \vee loves(x, Candy)$

2. Every child loves every candy.

$\forall x, \forall y : child(x) \wedge Candy(y) \rightarrow loves(x, y)$

clause form : $\neg child(x) \vee \neg Candy(y) \vee loves(x, y)$

3. Anyone who loves some candy is not a nutrition fanatic.

$\forall x, \exists y : Candy(y) \wedge loves(x, y) \rightarrow \neg fanatic(x)$

clause form : $\neg Candy(y) \vee \neg loves(x, y) \vee \neg fanatic(x)$

4. Anyone who eats any pumpkin is a nutrition fanatic.

$\forall x, \exists y : pumpkin(y) \wedge eats(x, y) \rightarrow fanatic(x)$

Clause form : $\neg pumpkin(y) \vee \neg eats(x, y) \vee fanatic(x)$

5. Anyone who buys any pumpkin either carves it or eats it.

$\forall x, \forall y : pumpkin(y) \wedge buy(x, y) \rightarrow carves(x, y) \vee eats(x, y)$

clause form : $\neg pumpkin(y) \vee \neg buy(x, y) \vee carves(x, y) \vee eats(x, y)$.

6. John buys a pumpkin.

$pumpkin(x) \wedge buy(John, x)$

7. Lifesavers is a candy.

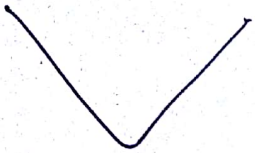
$Candy(Lifesaver)$.

Prove : $child(John) \rightarrow \exists x : pumpkin(x) \wedge carves(John, x)$.

[opposite] $pumpkin(x) \wedge carves(John, x) \rightarrow child(John)$

$\neg pumpkin(x) \vee \neg carves(John, x) \vee child(John)$.

(5) ¬ pumpkin(x) ∨ ¬ Carves(John, x) ∨ child(John)



(1) ¬ carves(John, x) ∨ child(John)

(6) ¬ Candy(y) ∨ love(x, y) ∨ ¬ carves(John, x)

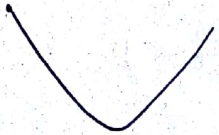
(2) love(John, lifesaver) ∨ carves(John, x)

¬ candy(lifesaver) ∨ ¬ fantic(lifesaver) ∨
carves(John, x

(3) ¬ fanatic(lifesaver) ∨ ¬ carves(John, x)

(4) ¬ pumpkin(y) ∨ ¬ eat(John, y) ∨ ¬ carve(John, y)

(5) ¬ pumpkin(y) ∨ ¬ buy(John, y)



□ NIL