Reasoning using First-Order Logic

Propositional Logic

- PL : Weak knowledge representation language
- Can not express complex AI problems
  - Hard to identify if the used entity is “individual” eg. Vishnu, Mumbai, 2021 etc
  - Can not directly represent properties of individual entities or relationships between the individual entities eg. Vishnu is tall
  - Can not express specialization, generalization or patterns eg. All rectangles have 4 sides

•**Predicate:** A predicate is a relationship that ties two atoms together in a sentence.

he following statement: "x is an integer." It has two parts: the first component, x, is the statement's subject, and the second half, "is an integer," is a

First Order Logic (FOL)

- **More expressive than PL**
- **Represents information using variables, relations and quantifiers**
- **Ex:** ***Krishna and Sudama are friends: => friends(Krishna, Sudama).***
- ***Tommy is a dog: => dog (Tommy).***

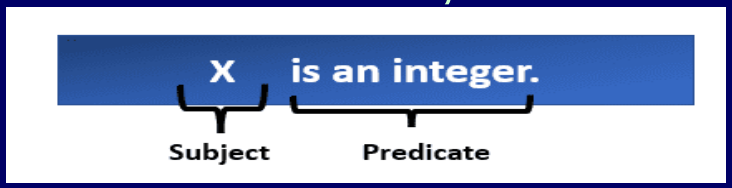
  today(Sunday)

Two types of first-order logic statements:

1. Subject: The major component of the sentence is the subject.

2. Predicate: A predicate is a relationship that ties two atoms together in a sentence.

Ex: "x is an integer." It has two parts: the first component, x, is the statement's subject, and the second half, "is an integer," is a predicate.



First Order Logic (FOL)

- **FOL symbols: Constant, Variable, Function**
- **Assuming 'x' is a domain of values,**
  - **Constant Term: A term with fixed value which belongs to the domain**
  - **Variable Term: A term which can be assigned values in the domain**
  - **Function: Say 'f' is a function of 'n' arguments. If we assume that t1,t2,t3,…,tn are terms then f (t1,t2,t3,…,tn) is also called as term.**
- **FOL: uses Propositional Logic as a base, so connectives are same**

First Order Logic (FOL)

- **FOL Rules:**
  - **Predicate Logic has two parts**
    a)Predecessor b) Successor
- **If predecessor is evaluated to TRUE, successor will be TRUE**
  - **Implication → symbol**
- **Eg. If the bag is of blue color, I will buy it.**
  - **Color(bag, blue) → buy(bag)**

First Order Logic (FOL)

- **Universal Quantifier '∀':**
- It specifies that the statement within its range is true for everything of every instance of particular thing
- "for all x" "for each x" "for every x"
  - ∀xA: A is True for every replacement of x
  - "All man drink coffee" : ∀x(man(x) --> drink(x, coffee))
  - "Every Gorilla is Black" : ∀x(Gorilla(x) → Black(x))
  - "Everyone at DJ is smart" : ∀x At (x, DJ) → smart(x)
  - "All birds fly" : ∀x(bird(x) --> fly(x))
  - "Every man respects his parent" :
    ∀xman(x) --> respects(x, parent)

First Order Logic (FOL)

- **Existential Quantifier** '∃': "there exists x" "for some x" "for atleast one x"
- Use AND or Conjunction symbol (∧) in Existential quantifiers
- Which express that the statement within its scope is true for at least one instance of something
  - ∃xA: A is True for at least one replacement of x
  - "There is a white dog" : ∃ x(Dog(x) ^ White(x))
  - "Some girls are intelligent" : ∃ x(girls(x) ^ Intelligent(x))
  - "Someone killed the cat and is guilty" :

    ∃ killed (x, cat) ^ guilty (x)

First Order Logic (FOL)

Properties of Quantifiers:

- In universal quantifier, ∀x∀y is similar to ∀y∀x.
- In Existential quantifier, ∃x∃y is similar to ∃y∃x.
- ∃x∀y is not similar to ∀y∃x.

- **Prof. Ram teaches either Maths or CS**

  Teaches(Ram, Maths) OR Teaches(Ram, CS)

- **Prof. Ram teaches Maths if and only if he does not teach CS**

  Teaches(Ram, Maths) ⇔ ¬ Teaches(Ram, CS)

- **All students like football**

  ∀x students x --> like (x, football)
- **Some students like football**

  ∃x students x ^ like (x, football)
- **Some students are happy**

  ∃x students x ^ happy (x)

Sentences

- A predicate is a sentence
- If sen, sen' are sentences & x a variable, then

  (sen), $\neg$sen, $\exists$x sen, $\forall$x sen,

  sen $\wedge$ sen', sen $\vee$ sen', sen $\Rightarrow$ sen'

  are sentences
- Nothing else is a sentence

Examples of Sentences

Birthday(x, y) – x celebrates birthday on date y

∀y ∃x Birthday (x, y) –

For all dates, there exists a person who celebrates his/her Birthday on that date.

That is – “everyday someone celebrates his/her Birthday”

Examples (contd.)

Brother(x, y) – x is y's brother
Loves (x, y) – x loves y

$\forall x \forall y$ Brother (x, y) $\Rightarrow$ Loves (x, y)
Everyone loves (all of) his/her brothers.

Let m (x) represent mother of x then
"everyone loves his/her mother" is

$\forall x$ Loves (x, m (x))

Examples (contd.)

1. Some boys play cricket.

In this question, the predicate is "play(x, y), " where x= boys, and y= game. Because there are some boys so we will use ∃, and it will be portrayed as:

∃x boys(x) ^ play(x, cricket).

2. Not all students like both Mathematics and Science.

The predicate in this question is "like(x, y)," where x= student, and y= subject. Because there are not all students, so we will use ∀ with negation, so following portray for this:

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

Quiz Revisited

- Some dogs bark
- $\exists x\ (dog(x) \wedge bark(x))$

- All dogs have four legs
- $\forall x\ (dog(x) \rightarrow have\_four\_legs\ (x)$
- $\forall x\ (dog(x) \rightarrow legs\ (x,4)$

- All barking dogs are irritating

  Do it yourself

- **No dogs purr**

  $\neg \exists x \ (dog(x) \wedge purr(x))$

- **Fathers are male parents with children**

  $\forall x \ (father(x) \rightarrow male(x) \wedge has\_children(x))$

- **Students are people who are enrolled in courses**

  Do it yourself

Examples (contd.)

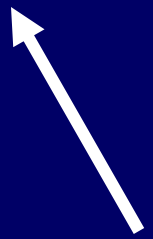
- Any number is the successor of its predecessor
- succ (x), pred (x),
  equal (x, y)

$$\forall x \text{ equal } (x, \text{succ } (\text{ pred } (x)))$$

Alternative Representation

- **The previous example can be represented as,**

$$\forall x\ (succ\ (\ pred\ (x\ ) = x)$$



Not Allowed in predicates

FOL with Equality

- In FOL with equality, we are allowed to use the equality sign (=) between two functions.
- This is just for representational ease
- We modify the definition of sentence to include equality as

  term = term is also a sentence

- Eg. "Manas has at least two brother"

  $\exists x, \exists y$ Brother(x, Manas) ^ Brother(y, Manas) ^ ¬ (x=y)

- NOTE: Above rule is TRUE under a given interpretation iff both terms refer to the same object

Practice problems

- Every rational number is a real number
- Some real numbers are rational numbers
- Not every real number is a rational number

Inference Rules

- Universal Generalization
- Universal Instantiation (Elimination)
- Existential Instantiation (Elimination)
- Existential Introduction

Inference Rules

- **Universal Generalization**
  - **It states that if premise p(c) is TRUE for any arbitrary element 'c' in the universe of discourse then we can have conclusion as ∀xp(x) is TRUE**
  - **It can be represented as p(c) / ∀xp(x)**
  - **E.g. p(c): "A byte contains 8 bit" if its TRUE then**

    ∀xp(x) : "All bytes contains 8 bits" will be TRUE

Inference Rules

- **Universal Instantiation (Elimination)**
  - **It can be applied multiple times to add a new sentence**
  - **It states that we can infer any sentence, premise p(c) by substituting a ground term c (a constant within domain x) for ∀xp(x) for any object in the universe of discourse**
  - **It can be represented as ∀xp(x) / p(c)**
  - **The substitution should be done by a constant term**

Inference Rules

- **Universal Instantiation (Elimination)**
  - **E.g.**

    ∀xp(x) : "Every person like Ice-cream"

    So we can infer that

    p(c): "Vishnu likes Ice-cream"
  - **E.g.**

∀x Likes (x, flower)

substituting x by Shirin gives

Likes (Shirin, flower)

Inference Rules

- **Existential Instantiation/Elimination (Skolemization)**
  - **It can be applied only once to replace the Existential sentence**
  - **It states that one can inter p(c) from the formula given in the form ∃xp(x) for a new constant symbol 'c'**
  - **It can be represented as ∃xp(x) / p(c)**
  - **E.g.**

∃ x Likes (x, flower) ⇒ Likes (shirin, flower)

as long as person is not in the knowledge base

Inference Rules

- Existential Introduction / Generalization
  - It states that if there is some element 'c' in the universe of discourse which has a property 'p', then we can infer that these there exists something in the universe which has property 'p'
  - E.g. Vishnu got good marks in Maths" : p(c)
  - Therefore "someone got good marks in Maths" : ∃xp(x)
  - It can be represented as p(c) /
  - Likes (Shahid, flower)

  can be written as

  ∃ x Likes (x, flower)

Forward Backward Chaining

- **Inference in FOL:**
- In a forward chaining system you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions) given those facts
- In a backward chaining system you start with some hypothesis (or goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new subgoals to prove as you go.

Forward Backward Chaining

- **Inference in FOL:**
- Data or Goal Driven?
  - Forward chaining systems are primarily data-driven
  - backward chaining systems are goal-driven

Forward Chaining

- Facts in the system are represented in a working memory which is continually updated.
- Rules in the system represent possible actions to take when specified conditions hold on items in the working memory
- They are sometimes called condition-action rules
- The conditions are usually patterns that must match items in the working memory

Forward Chaining

- actions usually involve adding or deleting items from the working memory.
- interpreter controls the application of the rules, given the working memory, thus controlling the system's activity.
- It is based on a cycle of activity sometimes known as a recognize-act cycle
- The system first checks to find all the rules whose conditions hold
- selects one and performs the actions in the action part of the rule
- selection of a rule to fire is based on fixed strategies, known as conflict resolution strategies

Forward Chaining

- actions usually involve adding or deleting items from the working memory.
- interpreter controls the application of the rules, given the working memory, thus controlling the system's activity.
- It is based on a cycle of activity sometimes known as a recognize-act cycle
- The system first checks to find all the rules whose conditions hold
- selects one and performs the actions in the action part of the rule
- selection of a rule to fire is based on fixed strategies, known as conflict resolution strategies

Forward Chaining

- Example: Consider simple example. Let's understand how the same example can be solved using both forward and backward chaining.
- Given facts:
- It is a crime for an American to sell weapons to the enemy of America.
- Country Nono is an enemy of America.
- Nono has some missiles.
- All the missiles were sold to Nono by Colonel West.
- Missile is a weapon.
- Colonel West is American.
- We have to prove that West is criminal.

Forward Chaining

- Let's see how to represent these facts by FOL.
- It is a crime for an American to sell weapons to the enemy nations.

American (x) ^Weapon(y) ^ sell (x.y.z) ^ enemy (z, America) => Criminal(x).

- Country Nono is an enemy of America.

  Enemy (Nono, America)
- Nono has some missiles.

  Owns (Nono, x)

  Missile(x)
- All the missiles were sold to Nono by Colonel West.

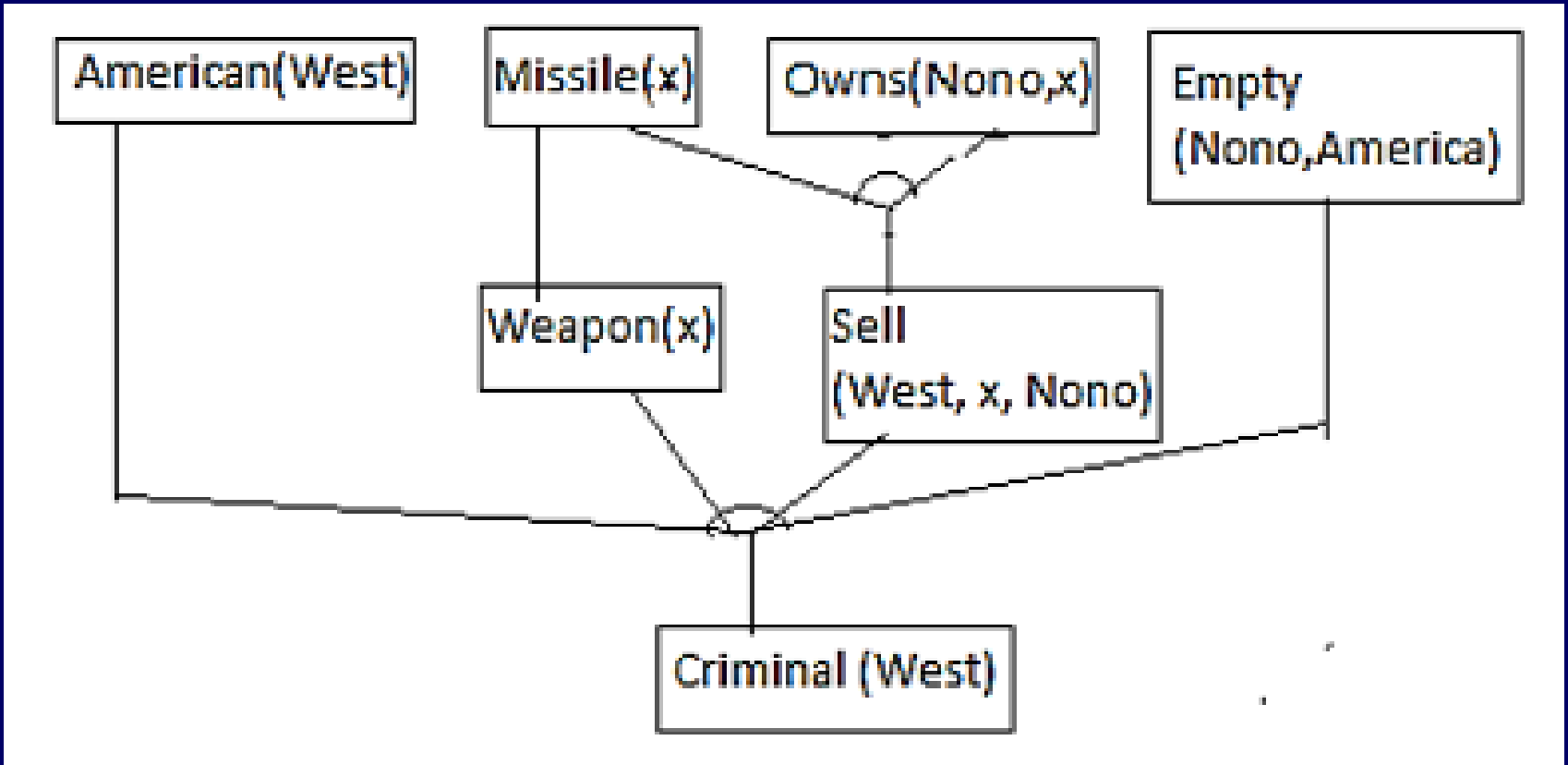
  Missile(x) ^ owns(Nono,x) => Sell(West,x, Nono)
- Missile is a weapon.

  Missile(x)=> weapon(x)
- Colonel West is American.

  American (West).

Forward Chaining

- Proof by forward chaining: The proof will start from the given facts. And we can derive other facts from those, it will lead us to the Solution. Answer observe from facts we can reach to the predicate Criminal (West).



Backward Chaining

- So far we have looked at how rule-based systems can be used to draw new conclusions from existing data, adding these conclusions to a working memory
- This approach is most useful when you know all the initial facts, but don't have much idea what the conclusion might be
- If you DO know what the conclusion might be, or have some specific hypothesis to test, forward chaining systems may be inefficient
- You COULD keep on forward chaining until no more rules apply or you have added your hypothesis to the working memory

Backward Chaining

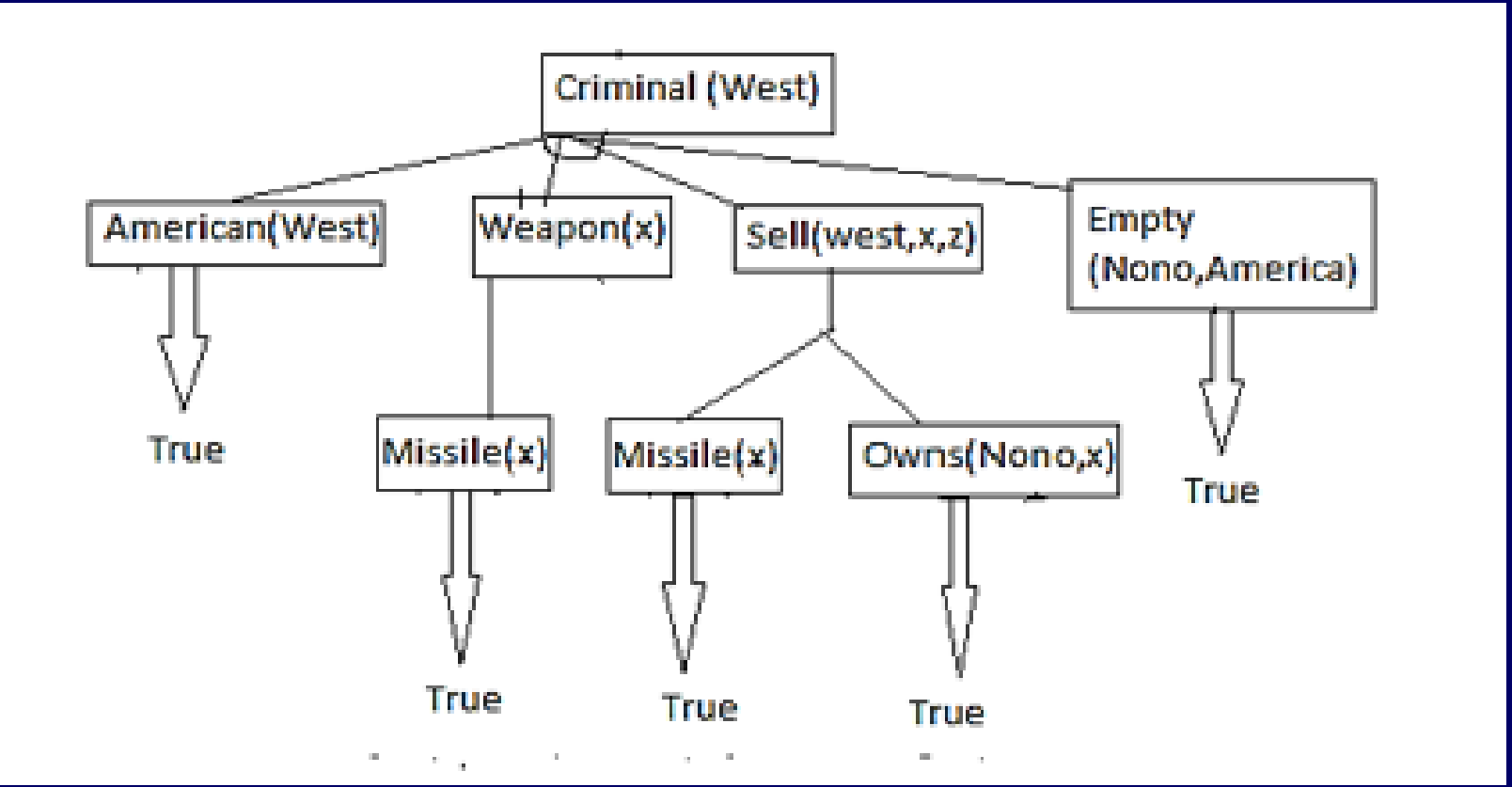
- But in the process the system is likely to do a lot of irrelevant work, adding uninteresting conclusions to working memory
- This can be done by backward chaining from the goal state (or on some hypothesized state that we are interested in)
- This is essentially what Prolog does, so it should be fairly familiar to you by now
- Given a goal state to try and prove (e.g., (bad-mood ali)) the system will first check to see if the goal matches the initial facts given

Backward Chaining

- If it does, then that goal succeeds
- If it doesn't the system will look for rules whose conclusions (previously referred to as actions) match the goal
- One such rule will be chosen, and the system will then try to prove any facts in the preconditions of the rule using the same procedure, setting these as new goals to prove
- Note that a backward chaining system does NOT need to update a working memory
- Instead it needs to keep track of what goals it needs to prove its main hypothesis.
- Lets take an example
-

Backward Chaining

- Proof by backward chaining: This proof will start from the fact to be proved. And we can map it with given facts, it will lead us to the solution. As from e.g. we observe all leaf nodes of proof are given facts that mean ”West is Criminal”.



Forward vs Backward Chaining

- Whether you use forward or backwards reasoning to solve a problem depends on the properties of your rule set and initial facts.
- Sometimes, if you have some particular goal (to test some hypothesis), then backward chaining will be much more efficient, as you avoid drawing conclusions from irrelevant facts.
- However, sometimes backward chaining can be very wasteful - there may be many possible ways of trying to prove something, and you may have to try almost all of them before you find one that works.
- Forward chaining may be better if you have lots of things you want to prove

Forward vs Backward Chaining

- when you have a small set of initial facts and when there tend to be lots of different rules which allow you to draw the same conclusion
- Backward chaining may be better if you are trying to prove a single fact, given a large set of initial facts, and where, if you used forward chaining, lots of rules would be eligible to fire in any cycle.
-

Reasoning in FOL

- Eg.
- Using Predicate logic find the course of Anish's liking for the following,
  I. Anish only likes easy course
  II. Computer courses are hard
  III. All electronic courses are easy
  IV. DSP is an electronics course

Reasoning in FOL

- **Using Predicate logic find the course of Anish's liking for the following,**

Solution: Converting given facts into FOL

I. ∀x : course(x)^easy(x) -> likes(Anish,x)

II. ∀x : course(x)^computer(x) -> hard(x)

III. ∀x : course(x)^electronics(x) -> easy(x)

IV. Eletronics(DSP)

V. Curse(DSP)

Reasoning in FOL

- **Consider the following problem:**

  If a perfect square is divisible by a prime p, then it is also divisible by square of p. Every perfect square is divisible by some prime.

  36 is a perfect square.

  Does there exist a prime q such that square of q divides 36?

Representation in FOL

- E.g.
- If a perfect square is divisible by a prime p, then it is also divisible by square of p.

$\forall x,y$ perfect_sq (x) $\wedge$ prime (y) $\wedge$ divides (x,y)
$\Rightarrow$ divides (x, square (y))

- Every perfect square is divisible by some prime.

$\forall x \exists y$ perfect_sq (x) $\wedge$ prime (y) $\wedge$
divides (x,y)

Representation in FOL

- **36 is a perfect square.**

  perfect_sq (36)

- **Does there exist a prime q such that the square of q divides 36?**

  $\exists$y prime (y) $\wedge$ divides (36,square(y))

The Knowledge base

1. $\forall$x,y perfect_sq (x) $\wedge$ prime (y) $\wedge$ divides (x,y) $\Rightarrow$ divides (x, square (y))
2. $\forall$x $\exists$y perfect_sq (x) $\wedge$ prime (y) $\wedge$ divides (x,y)
3. perfect_sq (36)

Inferencing

- From 2 and Universal Elimination

  (4) $\exists y$ perfect_sq (36) $\wedge$ prime (y) $\wedge$ divides (36, y)

- From 4 and Existential Elimination

  (5) perfect_sq (36) $\wedge$ prime (P) $\wedge$ divides (36, P)

- From (1) and (5)

  (6) divides (36, square(P))

Inferencing (contd.)

- From (5) and (6)

  (7) prime (P) $\wedge$ divides (36, square(P))

- From (7) and Existential Introduction

  $\exists$y prime (y) $\wedge$ divides (36, square(y))

Inferencing (contd.)

- However, inferencing would have been easier if the knowledge base consists of the following sentences
  - perfect_sq(36)
  - prime(P)
  - divides(36,P)
  - $\forall$x,y perfect_sq (x) $\wedge$ prime (y) $\wedge$ divides (x,y) $\Rightarrow$ divides (x, square (y))

Unification

- Unification is all about making the expressions look identical.
- The process of finding a substitution that makes two atomic sentences identical.
- To make them identical we need to do substitution.
  - UNIFY (Prime(7), Prime(x)) = {x/7}
  - UNIFY (Divides(49,x), Divides (y,7)) = {y/49, x/7}
  - UNIFY (Prime(7),Prime(17) – Impossible!
  - UNIFY (Divides(49,x), Divides (x,7)) – Impossible!

Unification

- Condition for Unification:
  - Predicate symbol must be same
  - Items of expression with different predicate symbol can never be unified
  - No. of arguments in both the expressions must be identical
  - Unification will fail if there are two similar variables present in same expressions.

Unification Algorithm

- **Unification Algorithm: Unify (L1,L2)**

Step 1:

IF L1 and L2 are variable or constant THEN

1) **IF L1 and L2 are identical THEN return NIL**
2) **ELSE IF L1 is a variable, THEN IF L1 occurs in L2 then return FAIL, ELSE return {L2/L1}**

– **return {L2/L1} means in place of L1 substitute L2**

3) ELSE IF L2 is a variable, THEN IF L2 occurs in L1 then return FAIL, ELSE return {L1/L2}

– **return {L1/L2} means in place of L2 substitute L1**

4) ELSE return FAIL

Unification Algorithm

- **Unification Algorithm: Unify (L1,L2)**

Step 2: IF the initial predicate symbol in L1 and L2 are not identical THEN return FAIL

Step 3: IF L1 and L2 have different no of arguments THEN return FAIL

Step 4: Set SUBST to NIL

Step 5: LOOP

{

For I <- 1 to no of arguments in L1

a) call unify with the ith argument of L1 and the ith argument of L2 putting result in S

b) if S= FAIL then return FAIL

Unification Algorithm

- **Unification Algorithm: Unify (L1,L2)**

c) if S!= NIL then

i) Apply S to the reminder of both L1 and L2

ii) SUBST = APPEND (S, SUBST)

}

Step 6: Return SUBST

NOTE: LOOP will perform actual substitution

Unification Examples

- Consider p (x, g(x))
- Sol:
  - P(z,y) : unifies with [x/z, g(x)/y]
    - Since both the terms are variables so both the expressions unifies
  - P(z,g(z)) : unifies with [x/z, g(z)/y]
    - Since both the terms are variables so both the expressions unifies
  - P(prime, f (prime)) : does not unifies as 'g' and 'f' does not match

Inference through Resolution in First Order Predicate Logic

Resolution – A technique of Inference

- It is a sound inference mechanism
- It is a theorem proving technique that proofs by **CONTRADICTION**
- It is used, if various statements are given and need to prove a conclusion of those statements
- Unification is a key concept in proofs by resolutions
- Resolution is a single inference rule which can efficiently operate on **C**onjunctive **N**ormal **F**orm (**CNF**) or **C**lausal **F**orm (**CF**)
- *Clause: Disjunction of literals is called clause*
- *CNF: A sentence represented as a conjunction of clauses said to be CNF*

Resolution – Steps

1. Conversion of facts into FOL
2. Convert FOL statements into CNF
3. Negate the statements which needs to prove (by contradiction)
4. Draw resolution graph (Unification)

Resolution – Example

1. John likes all kind of food
2. Apple and vegetable are good food
3. Anything anyone eats and not killed is food
4. Shirin eats peanuts and still alive
5. Shanon eats everything that Shirin eats

Prove by resolution that "John likes peanuts"

Resolution – A technique of Inference

- A sound inference mechanism

Principle:
Suppose x is a literal and S1 and S2 are two sets of propositional sentences represented n clausal form

If we have (x V S1) AND (¬x V S2)
Then we get S1 V S2
Here S1 V S2 is the resolvent,
x is resolved upon

Proof by refutation

1. Convert facts into First order logic (FOL)
2. Convert FOL into CNF (Conjuctive Normal Form)
3. Negate the statement to be proved , and add the result to the knowledge base.
4. Draw Resolution graph .
5. If empty clause (NIL) is produced , stop and report that original theorem is true .

Proof by refutation
Ex 1

1. If It is sunny and warm day you will enjoy
2. If it is raining you will get wet
3. It is warm day
4. It is raining
5. It is sunny

Goal: You will enjoy

Prove: enjoy

Proof by refutation
Ex 1

Step 1 : Conversion to first order logic

- If It is sunny and warm day you will enjoy
  Sunny ∧ warm → enjoy
- If it is raining you will get wet
  raining → wet
- It is warm day
  warm
- It is raining
  raining
- It is sunny
  Sunny

Proof by refutation
Ex 1

Step 2 : conversion to CNF

- **Sunny ∧ Warm → enjoy**

  Eliminate implication:

  ¬(Sunny ∧ warm) V enjoy

  Moving negation inside

  ¬Sunny V ¬warm V enjoy
- **raining → wet**

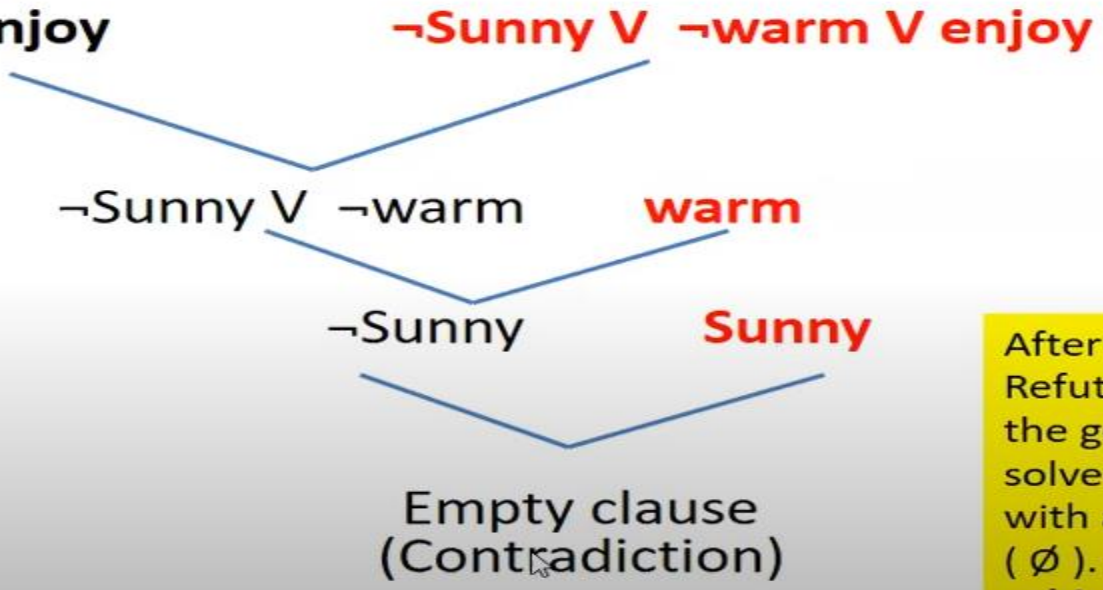
  Eliminate implication:

  ¬raining V wet
- **warm**
- **raining**
- **Sunny**

Proof by refutation
Ex 1

Step: 3 & 4 Resolution graph

- Negate the statement to be proved: **¬enjoy**
- Now take the statements one by one and create resolution graph.



After applying Proof by Refutation (Contradiction) on the goal, the problem is solved, and it has terminated with a **Null clause** (Ø). **Hence, the goal is achieved.**

Proof by refutation
Ex 2

- Consider the following Knowledge Base:

1. The humidity is high or the sky is cloudy.
2. If the sky is cloudy, then it will rain.
3. If the humidity is high, then it is hot.
4. It is not hot.

Goal: It will rain.

Proof by refutation
Ex 2

Step 1 : Conversion to first order logic

1. **The humidity is high or the sky is cloudy.**
2. If the sky is cloudy, then it will rain.
3. If the humidity is high, then it is hot.
4. It is not hot.

- **Let P : The humidity is high**
- **Let Q: sky is cloudy**

The humidity is high or the sky is cloudy

P V Q

Proof by refutation
Ex 2

Convert it to CNF

- P V Q
- Q → R
- P → S
- ¬S

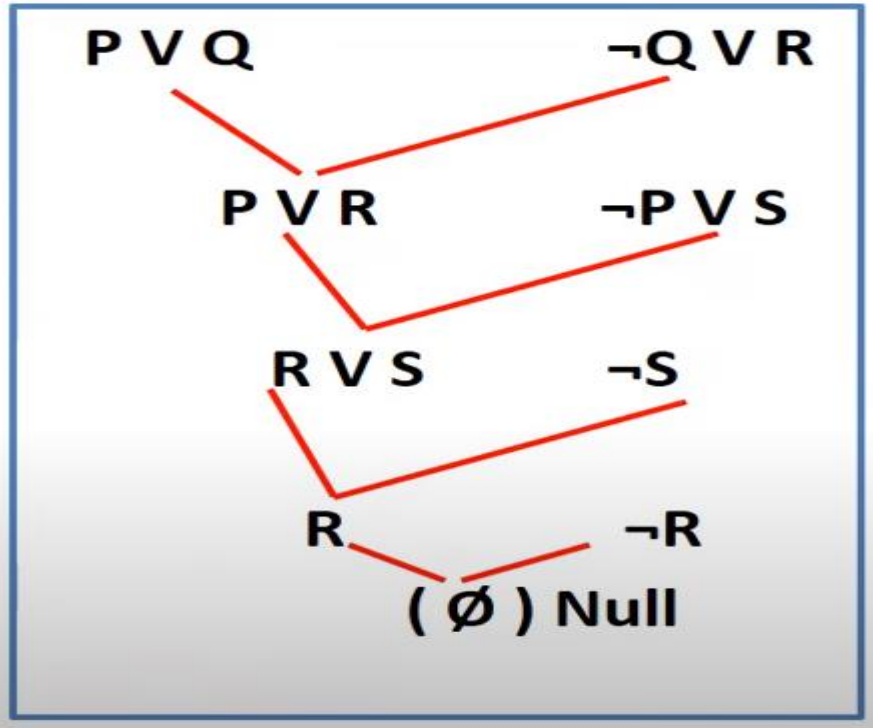
- P V Q
- ¬Q V R
- ¬P V S
- ¬S

Negation of Goal (¬R): It will not rain.

Proof by refutation
Ex 2

Resolution graph

- P V Q
- ¬Q V R
- ¬P V S
- ¬S
- ¬R (Goal)



Proof by refutation
Ex 3

1. Sunil likes all kind of food.
2. Apple and vegetable are food
3. Anything anyone eats and not killed is food.
4. Anil eats peanuts and still alive
5. Sohan eats everything that Anil eats.

- **Prove by resolution that:**

  Sunil likes peanuts

Proof by refutation
Ex 3

Conversion of Facts/Statements into FOL

1. Sunil likes all kind of food.
2. Apple and vegetable are food
3. Anything anyone eats and not killed is food.
4. Anil eats peanuts and still alive
5. Sohan eats everything that Anil eats
6. Sunil likes peanuts.

1. $\forall x : food(x) \rightarrow likes(Sunil, x)$
2. $food(apple) \wedge food(vegetable)$
3. $\forall x \forall y : eats(x , y) \wedge \neg killed(x) \rightarrow food(y)$
4. $eats(Anil , Peanut) \wedge alive(Anil)$
5. $\forall x: eats(Anil, x) \rightarrow eats(Sohan , x)$
6. $Likes(Sunil, Peanut)$
   - $\forall x : \neg killed(x) \rightarrow alive(x)$
   - $\forall x : alive(x) \rightarrow \neg killed(x)$

(added predicates)

Proof by refutation
Ex 3

Conversion of FOL into CNF: Elimination of →

1. ∀x : food(x)→likes(Sunil, x)
2. food(apple) ∧ food(vegetable)
3. ∀x ∀y : eats(x , y) ∧ ¬ killed(x)→food(y)
4. eats(Anil , Peanut) ∧ alive(Anil)
5. ∀x: eats(Anil, x) → eats(Sohan , x)
6. ∀x : ¬ killed(x)→alive(x)
7. ∀x : alive (x)→¬killed (x)
8. Likes(Sunil, Peanut)

1. ∀x ¬ food(x) V likes(Sunil, x)
2. food(Apple) ∧ food(vegetables)
3. ∀x ∀y ¬ [eats(x, y) ∧ ¬ killed(x)] V food(y)
4. eats (Anil, Peanuts) ∧ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Sohan, x)
6. ∀x¬ [¬ killed(x)] V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(Sunil, Peanuts)

Proof by refutation
Ex 3

Rename variables or standardize variables

1. ∀x ¬ food(x) V likes(Sunil, x)
2. food(Apple) ∧ food(vegetables)
3. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
4. eats (Anil, Peanuts) ∧ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Sohan, x)
6. ∀x killed(x) V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(Sunil, Peanuts)

1. ∀x ¬ food(x) V likes(Sunil, x)
2. food(Apple) ∧ food(vegetables)
3. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
4. eats (Anil, Peanuts) ∧ alive(Anil)
5. ∀w¬ eats(Anil, w) V eats(Sohan, w)
6. ∀g killed(g) V alive(g)
7. ∀k ¬ alive(k) V ¬ killed(k)
8. likes(Sunil, Peanuts)

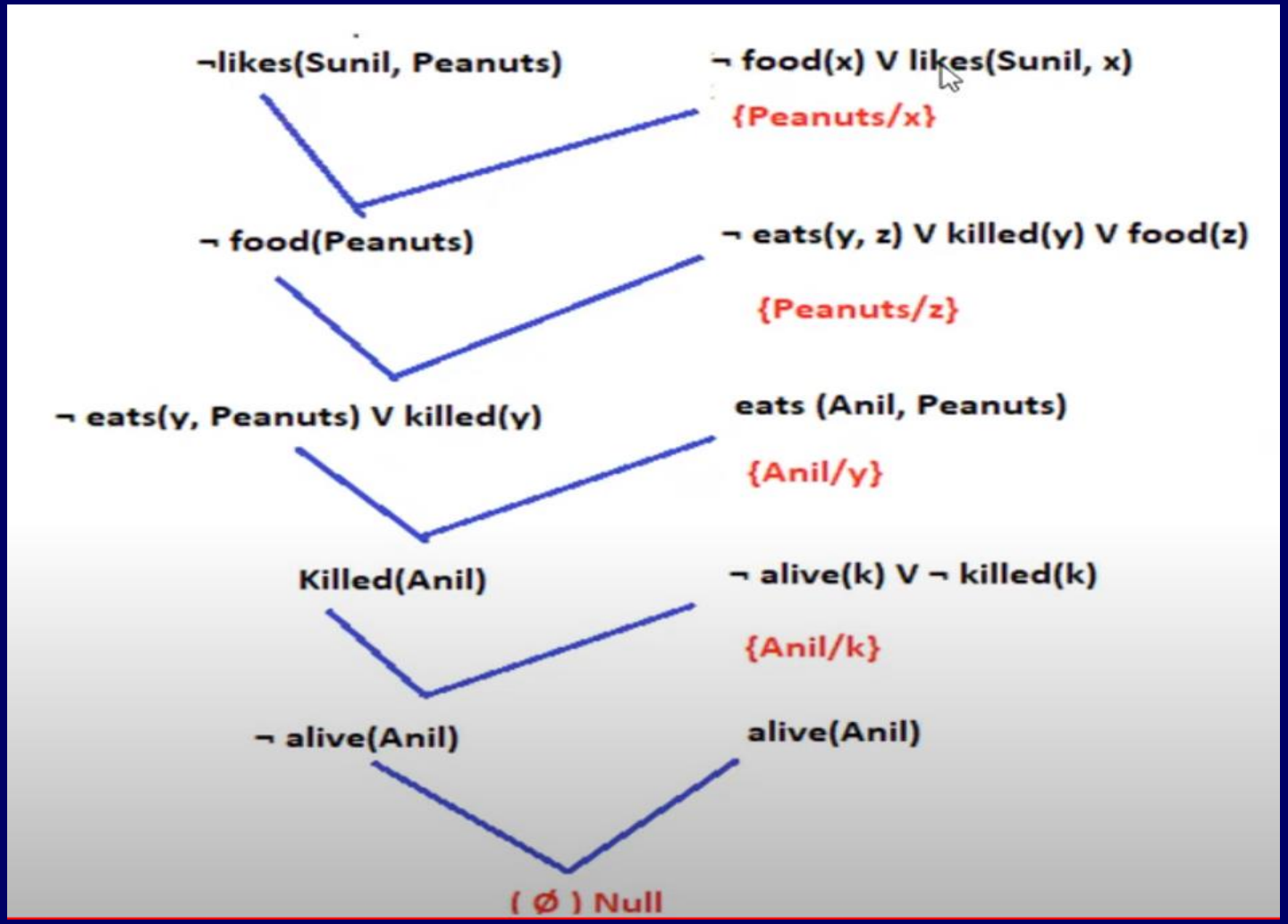
Proof by refutation
Ex 3

Drop Universal quantifiers

1. ∀x ¬ food(x) V likes(Sunil, x)
2. food(Apple) ∧ food(vegetables)
3. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
4. eats (Anil, Peanuts) ∧ alive(Anil)
5. ∀w¬ eats(Anil, w) V eats(Sohan, w)
6. ∀g killed(g) V alive(g)
7. ∀k ¬ alive(k) V ¬ killed(k)
8. likes(Sunil, Peanuts).

1. ¬ food(x) V likes(Sunil, x)
2. food(Apple)
3. food(vegetables)
4. ¬ eats(y, z) V killed(y) V food(z)
5. eats (Anil, Peanuts)
6. alive(Anil)
7. ¬ eats(Anil, w) V eats(Sohan, w)
8. killed(g) V alive(g)
9. ¬ alive(k) V ¬ killed(k)
10. likes(Sunil, Peanuts)

Proof by refutation
Ex 3
Resolution Graph



An Example

- If a triangle is equilateral then it is isosceles
- If a triangle is isosceles then two sides AB and AC are equal
- If AB and AC are equal then angle B and angle C are equal
- ABC is an equilateral triangle

- Angle B is equal to angle C -- Prove

- If a triangle is equilateral then it is isosceles

  Equilateral (ABC) → Isosceles (ABC)

- If a triangle is isosceles then two sides AB and AC are equal

  Isosceles (ABC) → Equal (AB,AC)

- If AB and AC are equal then angle B and angle C are equal

  Equal (AB,AC) → Equal (B,C)

- ABC is an equilateral triangle

  Equilateral (ABC)

- **Clausal Form**

1. Equilateral (ABC) → Isosceles (ABC)

¬ Equilateral (ABC) V Isosceles (ABC)

2. Isosceles (ABC) → Equal (AB,AC)

¬ Isosceles (ABC) V Equal (AB,AC)

3. Equal (AB,AC) → Equal (B,C)

¬Equal (AB,AC) → Equal (B,C)

4. Equilateral (ABC)

Proof by refutation

- To prove

Angle B is equal to Angle C

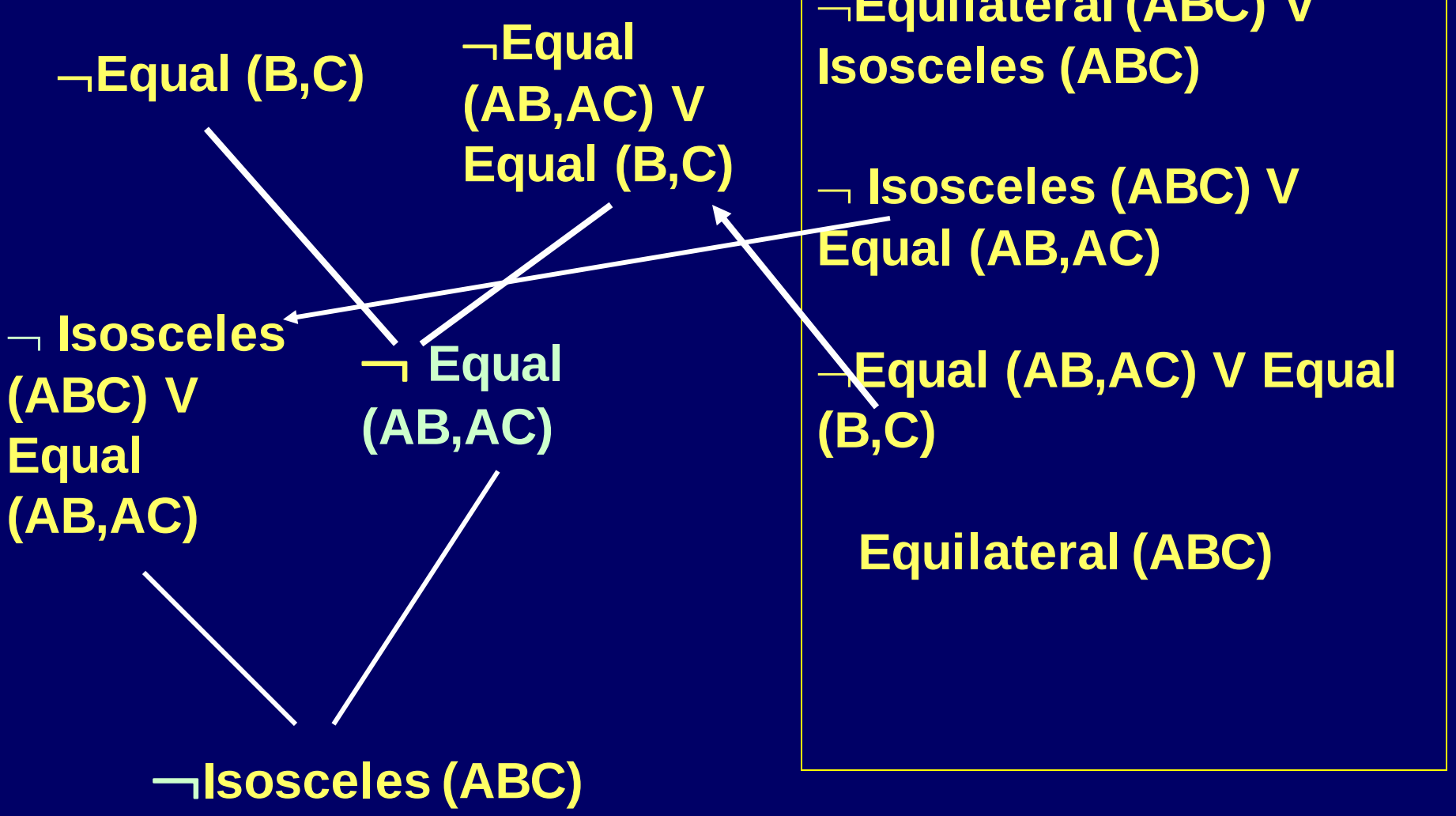
Equal (B,C)

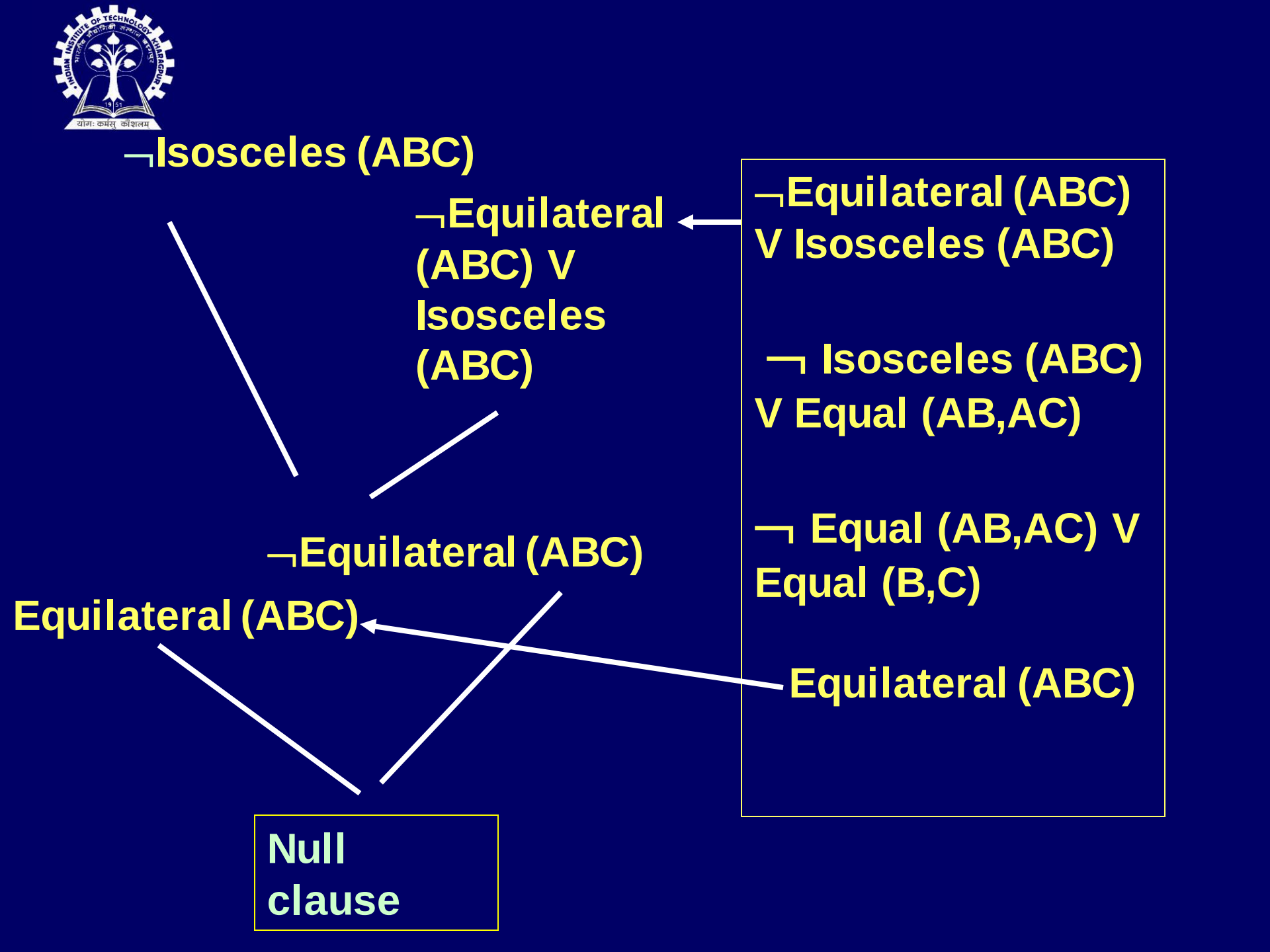
Let us disprove

Not equal (B,C)

¬Equal (B,C)

Let us try to disprove this


¬Equal (B,C)
¬Equal (AB,AC) V Equal (B,C)
¬Equilateral (ABC) V Isosceles (ABC)
¬ Isosceles (ABC) V Equal (AB,AC)
¬ Isosceles (ABC) V Equal (AB,AC)
¬ Equal (AB,AC)
¬Equal (AB,AC) V Equal (B,C)
Equilateral (ABC)
¬Isosceles (ABC)


¬Isosceles (ABC)
¬Equilateral (ABC) V Isosceles (ABC)
¬Equilateral (ABC) V Isosceles (ABC)
¬ Isosceles (ABC) V Equal (AB,AC)
¬ Equal (AB,AC) V Equal (B,C)
¬Equilateral (ABC)
Equilateral (ABC)
Equilateral (ABC)
Null clause

Procedure for Resolution

- Convert given propositions into clausal form
- Convert the negation of the sentence to be proved into clausal form
- Combine the clauses into a set
- Iteratively apply resolution to the set and add the resolvent to the set
- Continue until no further resolvents can be obtained or a null clause is obtained

What is it for Predicate Logic

A Few Statements

- All people who are graduating are happy.
- All happy people smile.
- Someone is graduating.
- Is someone smiling? (Conclusion)

Solving the problem

- We intend to code the problem in predicate calculus.
- Use resolution refutation to solve problem
- Solving ≈ whether the conclusion can be answered from the given set of sentences.

Selecting the Predicates

- graduating (x) : x is graduating
- Happy (x) : x is happy
- Smiling (x) : x is smiling

Encoding sentences in Predicate Logic

- All people who are graduating are happy
  - $\forall(x)\,[\text{graduating}\,(x) \rightarrow \text{happy}\,(x)]$
- All happy people smile
  - $\forall(x)\,[\text{happy}\,(x) \rightarrow \text{smiling}\,(x)]$
- Someone is graduating
  - $\exists(x)\,\text{graduating}\,(x)$
- Is someone smiling
  - $\exists(x)\,\text{smiling}\,(x)$

Predicates

1. $\forall$(x) graduating (x) $\rightarrow$ happy (x)

2. $\forall$(x) happy (x) $\rightarrow$ smiling (x)

3. $\exists$(x) graduating (x)

4. $\neg \exists$(x) smiling (x)
   (Negating the conclusion)

Converting to Clausal form

- Step 1: Eliminate $\rightarrow$

1. $\forall(x) \neg$ **graduating (x)** $\vee$ **happy (x)**
2. $\forall(x) \neg$ **happy (x)** $\vee$ **smiling (x)**
3. $\exists(x)$ **graduating (x)**
4. $\neg \exists(x)$ **smiling (x)**

Converting to Canonical form

- Step 2: Reduce the scope of negation

1. $\forall(x) \neg$ **graduating (x)** $\vee$ **happy (x)**
2. $\forall(x) \neg$ **happy (x)** $\vee$ **smiling (x)**
3. $\exists(x)$ **graduating (x)**
4. $\forall(x) \neg$ **smiling (x)**

Converting to Canonical form

- Step 3: Standardize variables apart

1. $\forall(x) \neg$ **graduating (x)** $\vee$ **happy (x)**
2. $\forall(y) \neg$ **happy (y)** $\vee$ **smiling (y)**
3. $\exists(z)$ **graduating (z)**
4. $\forall(w) \neg$ **smiling (w)**

Converting to Canonical form

- Step 4: Move all quantifiers to the left

1. $\forall(x) \neg$ **graduating (x)** $\vee$ **happy (x)**
2. $\forall(y) \neg$ **happy (y)** $\vee$ **smiling (y)**
3. $\exists(z)$ **graduating (z)**
4. $\forall(w) \neg$ **smiling (w)**

Converting to Canonical form

- Step 5: Eliminate ∃ (Skolemization)

1. **∀(x) ¬ graduating (x) ∨ happy (x)**
2. **∀(y) ¬ happy (y) ∨ smiling (y)**
3. **graduating (name1)**
   (name1 is the skolemization constant)
4. **∀(w) ¬ smiling (w)**

Converting to Canonical form

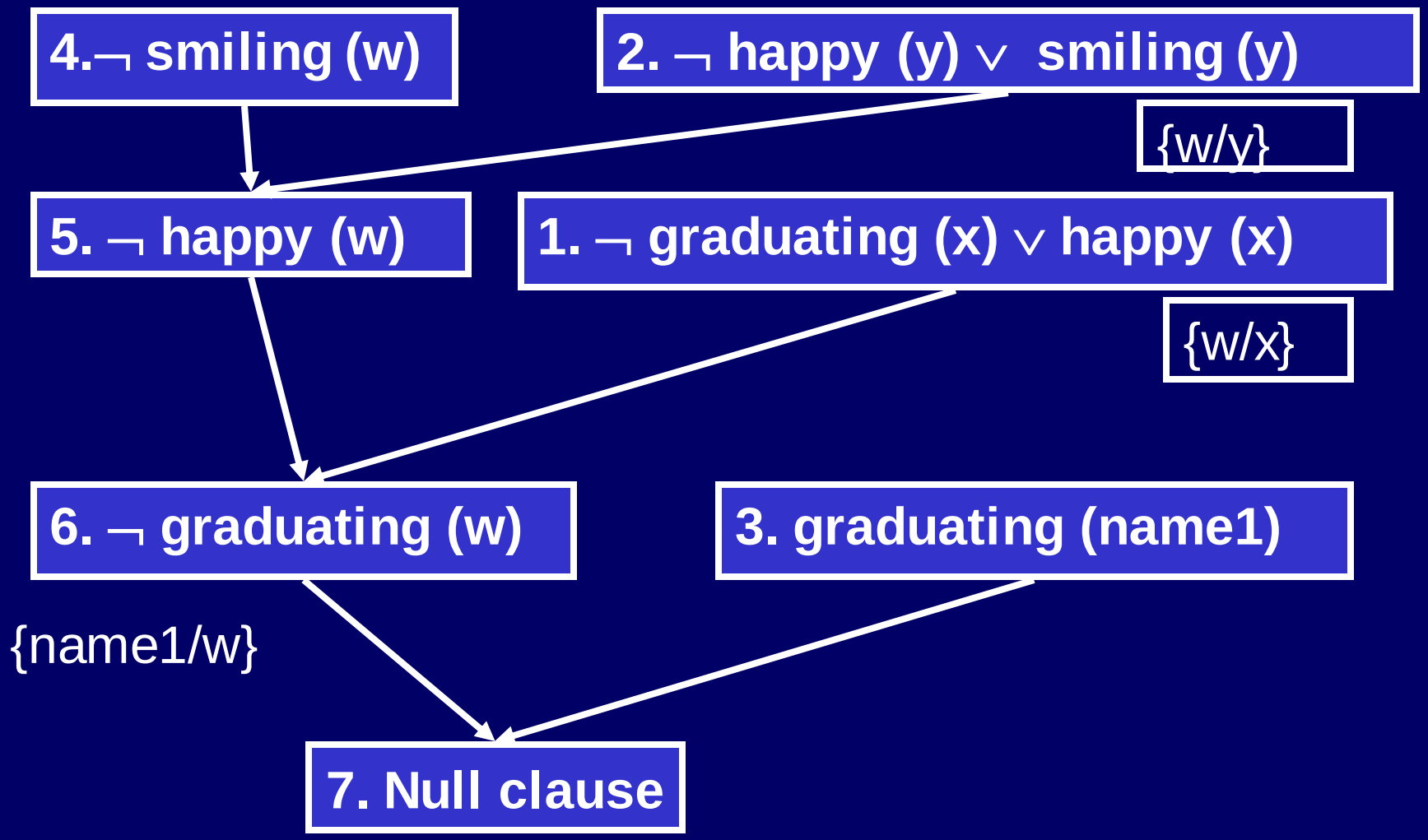
- Step 6. Drop all $\forall$

1. $\neg$ **graduating (x)** $\vee$ **happy (x)**
2. $\neg$ **happy (y)** $\vee$ **smiling (y)**
3. **graduating (name1)**
4. $\neg$ **smiling (w)**

Converting to Canonical form

- Step 7. Convert to conjunct of disjuncts form
- Step 8. Make each conjunct a separate clause.
- Step 9. Standardize variables apart again.
- **These steps do not change the set of clauses any further (in the present problem)**

Resolution


4. ¬ smiling (w)
2. ¬ happy (y) ∨ smiling (y)
{w/y}
5. ¬ happy (w)
1. ¬ graduating (x) ∨ happy (x)
{w/x}
6. ¬ graduating (w)
3. graduating (name1)
{name1/w}
7. Null clause

Example

1. Sunil likes all kind of food.
2. Apple and vegetable are food
3. Anything anyone eats and not killed is food.
4. Anil eats peanuts and still alive
5. Sohan eats everything that Anil eats.

- Prove by resolution that:

  Sunil likes peanuts

Example

Drop Universal quantifiers

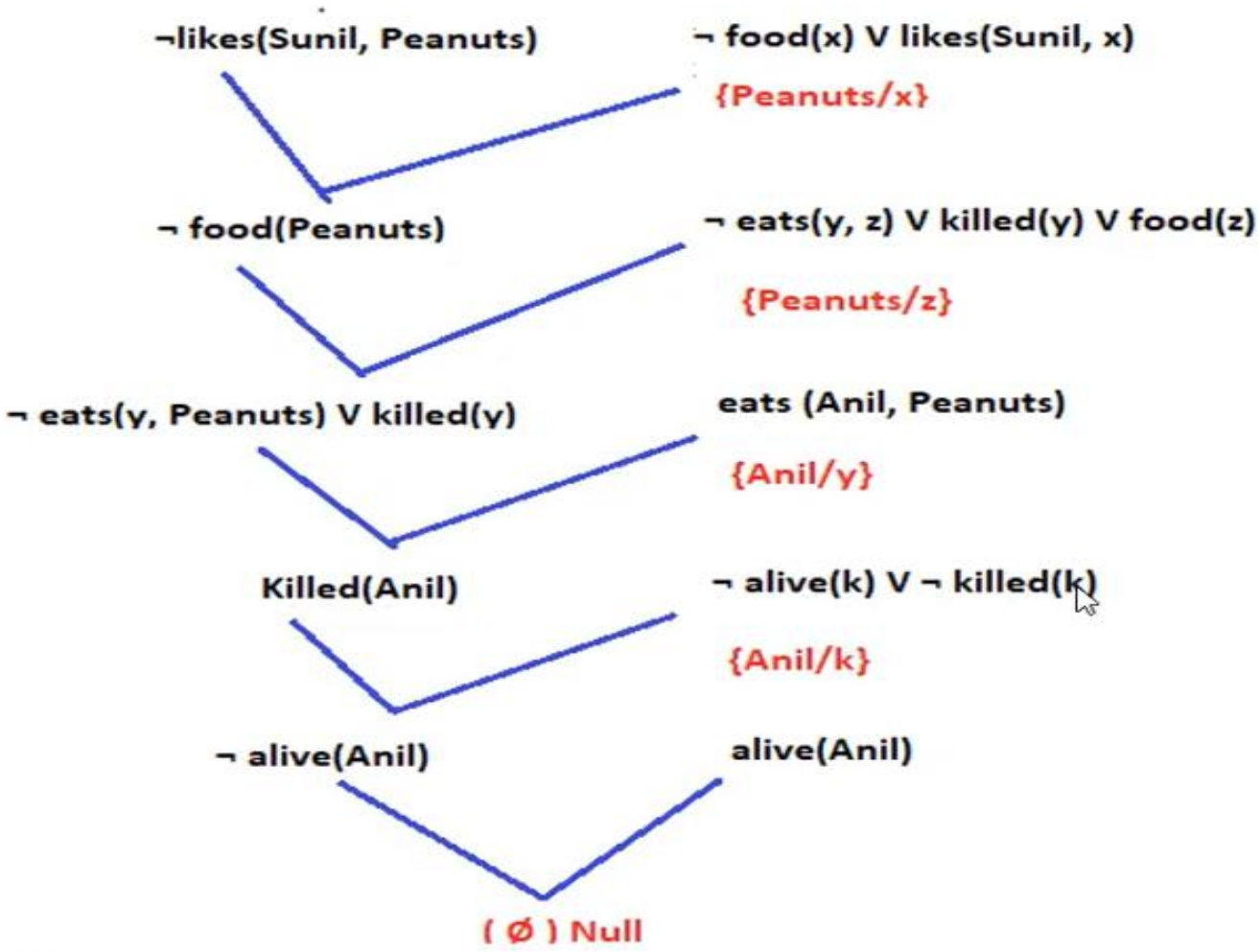
1. ∀x ¬ food(x) V likes(Sunil, x)
2. food(Apple) Λ food(vegetables)
3. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
4. eats (Anil, Peanuts) Λ alive(Anil)
5. ∀w¬ eats(Anil, w) V eats(Sohan, w)
6. ∀g killed(g) V alive(g)
7. ∀k ¬ alive(k) V ¬ killed(k)
8. likes(Sunil, Peanuts).

1. ¬ food(x) V likes(Sunil, x)
2. food(Apple)
3. food(vegetables)
4. ¬ eats(y, z) V killed(y) V food(z)
5. eats (Anil, Peanuts)
6. alive(Anil)
7. ¬ eats(Anil, w) V eats(Sohan, w)
8. killed(g) V alive(g)
9. ¬ alive(k) V ¬ killed(k)
10. likes(Sunil, Peanuts)

Statements "food(Apple) Λ food(vegetables)" and "eats (Anil, Peanuts) Λ alive(Anil)" can be written in two separate statements.

Example

Resolution graph



Quiz

Solve the problem with resolution

If a perfect square is divisible by a prime p, then it is also divisible by square of p. Every perfect square is divisible by some prime.

36 is a perfect square.

Does there exist a prime q such that square of q divides 36?

Answer Extraction

Find the Package Example (Nilsson)

- We know that
- All packages in room 27 are smaller than those in room 28
- Package A is either in room 27 or in room 28
- Package B is in room 27
- Package B is not smaller than Package A
- Where is Package A?

Answer extraction

- Suppose we wish to prove whether KB |= (∃u)f(u)
- We are probably interested in knowing the w for which f(u) holds.
- Add Ans(u) literal to each clause coming from the negation of the theorem to be proven; stop resolution process when there is a clause containing only Ans literal

1. All packages in Room 27 are smaller than those in room 28

$\neg P(x) \vee \neg P(y) \vee \neg I(x, 27) \vee \neg I(y, 28) \vee S(x,y)$

2. $P(A)$

3. $P(B)$

4. $I(A,27) \vee I(A,28)$

5. $I(B,27)$

6. $\neg S(B,A)$

In which room is A? That is, Prove $(\exists u)\, I(A, u)$

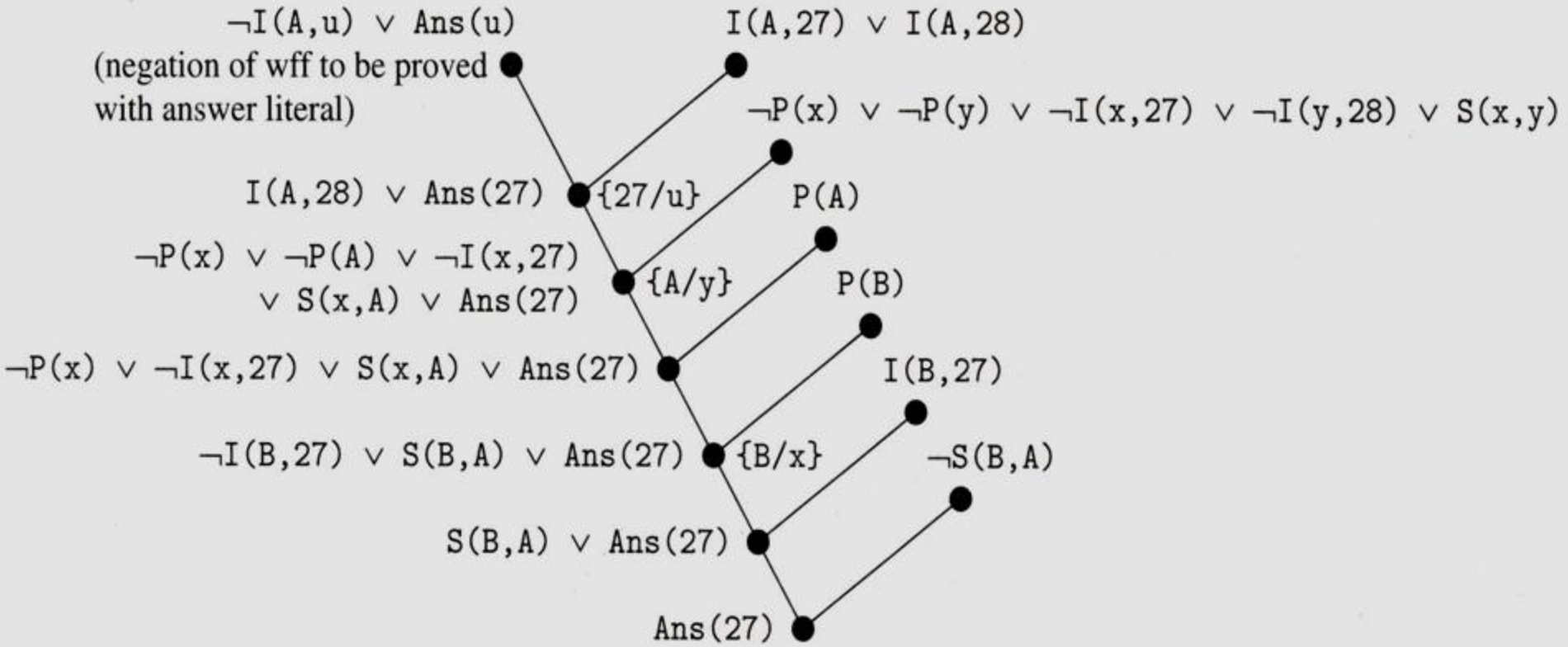

¬I(A,u) ∨ Ans(u)
(negation of wff to be proved
with answer literal)
I(A,27) ∨ I(A,28)
¬P(x) ∨ ¬P(y) ∨ ¬I(x,27) ∨ ¬I(y,28) ∨ S(x,y)
I(A,28) ∨ Ans(27)
{27/u}
P(A)
¬P(x) ∨ ¬P(A) ∨ ¬I(x,27)
∨ S(x,A) ∨ Ans(27)
{A/y}
P(B)
¬P(x) ∨ ¬I(x,27) ∨ S(x,A) ∨ Ans(27)
I(B,27)
¬I(B,27) ∨ S(B,A) ∨ Ans(27)
{B/x}
¬S(B,A)
S(B,A) ∨ Ans(27)
Ans(27)

Figure 16.2

Prolog: Family Tree

- Prolog is a language built around the Logical Paradigm: a declarative approach to problem-solving.
- There are only three basic constructs in Prolog: **facts**, **rules**, and **queries**.
- **knowledge base** (or a database): A collection of facts and rules is called
- Prolog programs simply are knowledge bases, collections of facts and rules which describe some collection of relationships that we find interesting.
- So how do we use a Prolog program?
- By posing **queries**. That is, by asking questions about the information stored in the knowledge base. The computer will automatically find the answer (either True or False) to our queries.

Prolog: Family Tree

- Relationship is one of the main features that we have to properly mention in Prolog.
- These relationships can be expressed as facts and rules.
- After that we will see about the family relationships, how we can express family based relationships in Prolog, and also see the recursive relationships of the family.
- We will create the knowledge base by creating facts and rules, and play query on them.

Prolog: Family Tree

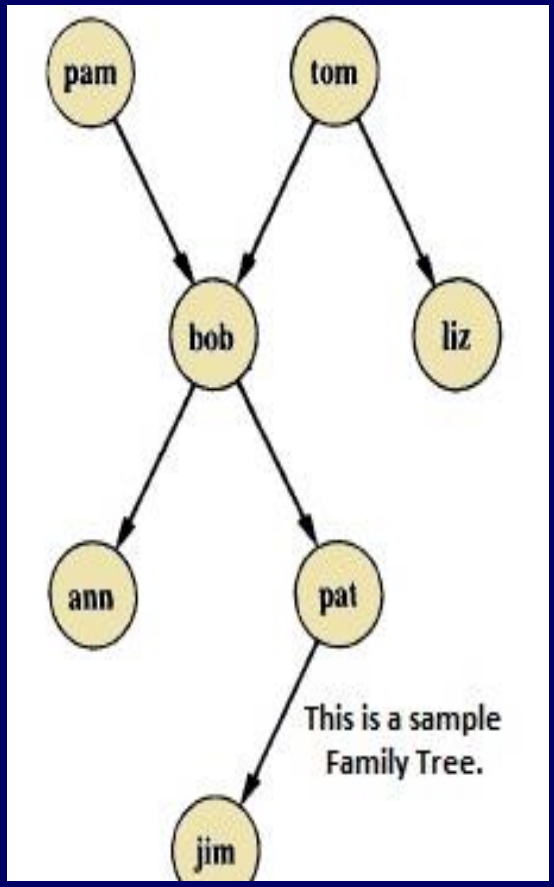
- In Prolog programs, it specifies relationship between objects and properties of the objects.
- There are various kinds of relationships, of which some can be rules as well. A rule can find out about a relationship even if the relationship is not defined explicitly as a fact.
- We can define a brother relationship as follows −
- Two person are brothers, if,
  - They both are male.
  - They have the same parent.

Prolog: Family Tree

- Now consider we have the below phrases −
  - parent(sudip, piyus).
  - parent(sudip, raj).
  - male(piyus).
  - male(raj).
  - brother(X,Y) :- parent(Z,X), parent(Z,Y),male(X), male(Y)
  - ➢ These clauses can give us the answer that piyus and raj are brothers, but we will get three pairs of output here. They are: (piyus, piyus), (piyus, raj), (raj, raj). For these pairs, given conditions are true, but for the pairs (piyus, piyus), (raj, raj), they are not actually brothers, they are the same persons. So we have to create the clauses properly to form a relationship.
  - ➢ The revised relationship can be as follows −
  - A and B are brothers if −
  - A and B, both are male
  - They have same father
  - They have same mother
  - A and B are not same

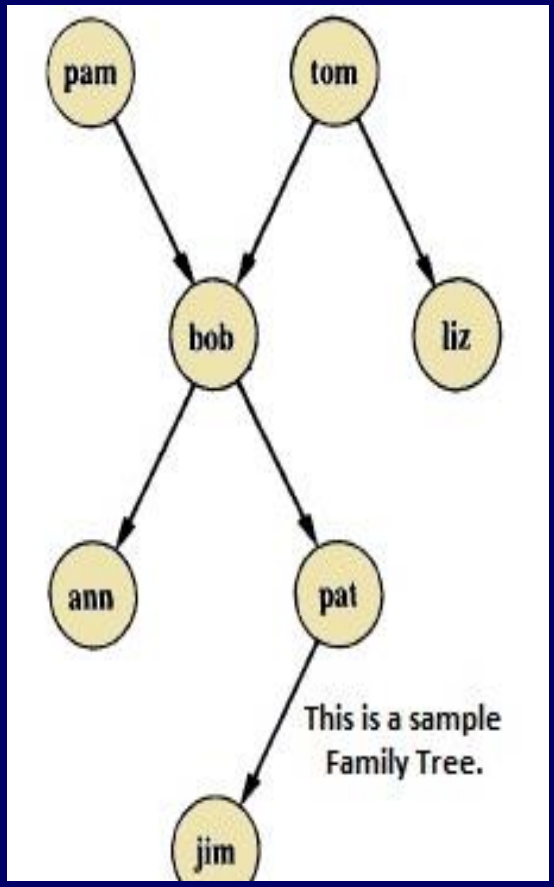
Prolog: Family Tree

- Family Relationship in Prolog
  - Here we will see the family relationship. This is an example of complex relationship that can be formed using Prolog. We want to make a family tree, and that will be mapped into facts and rules, then we can run some queries on them.
- Suppose the family tree is as follows
  - Here from this tree, we can understand that there are few relationships. Here bob is a child of pam and tom, and bob also has two children — ann and pat. Bob has one brother liz, whose parent is also tom.



Prolog: Family Tree

- Family Relationship in Prolog
  - So we want to make predicates as follows ,
  - Predicates
  - parent(pam, bob).
  - parent(tom, bob).
  - parent(tom, liz).
  - parent(bob, ann).
  - parent(bob, pat).
  - parent(pat, jim).
  - parent(bob, peter).
  - parent(peter, jim).



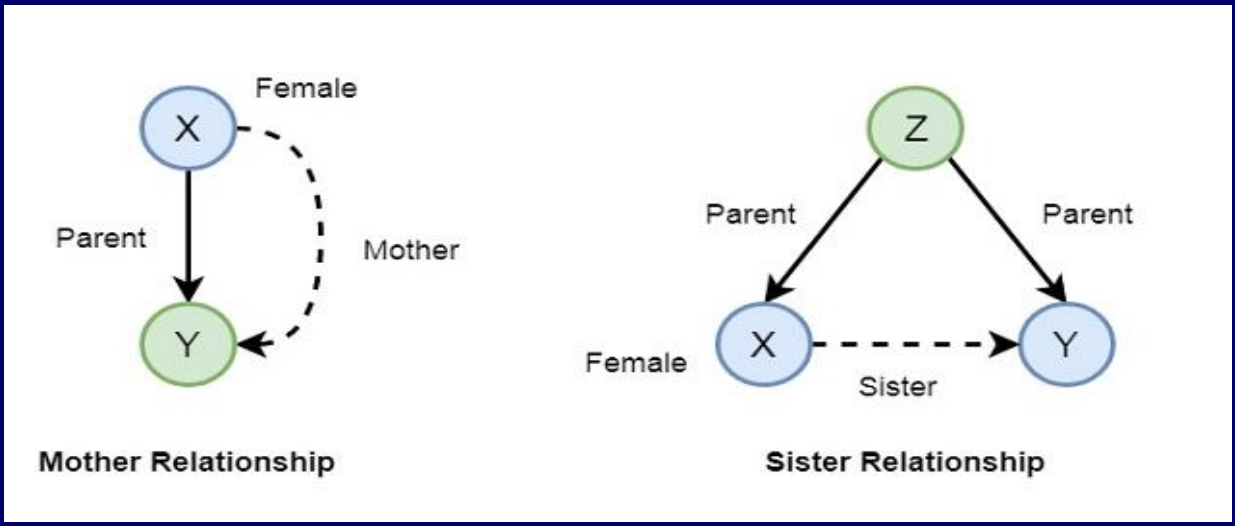
Prolog: Family Tree

- Family Relationship in Prolog
  - Some facts can be written in two different ways, like sex of family members can be written in either of the forms ,
  - Predicates
- female(pam).
- male(tom).
- male(bob).
- female(liz).
- female(pat).
- female(ann).
- male(jim).
- Or in the below form −
- sex(pam, feminine).
- sex(tom, masculine).
- sex(bob, masculine)…. and so on.

Prolog: Family Tree

A Prolog program consists of clauses terminated by a full stop.

- The arguments of relations can (among other things) be:
- Concrete objects, or constants (such as pat and jim),
- General objects such as X and Y.
- Objects of the first kind in our program are called atoms.
- Objects of the second kind are called variables.
- Questions to the system consist of one or more goals.
- In Prolog syntax, we can write −
  - mother(X,Y) :- parent(X,Y), female(X).
  - sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X), X \== Y.



Prolog: Family Tree

Knowledge Base (family.pl)

```
female(pam).
female(liz).
female(pat).
female(ann).
male(jim).
male(bob).
male(tom).
male(peter).
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
parent(bob,peter).
parent(peter,jim).
mother(X,Y):- parent(X,Y),female(X).
father(X,Y):- parent(X,Y),male(X).
haschild(X):- parent(X,_).
```

Prolog: Family Tree

- Output

```
| ?- [family].
compiling D:/TP Prolog/Sample_Codes/family.pl for byte code...
D:/TP Prolog/Sample_Codes/family.pl compiled, 23 lines read - 3088 bytes written, 9 ms

yes
| ?- parent(X,jim).

X = pat ? ;

X = peter

yes
| ?-
mother(X,Y).

X = pam
Y = bob ? ;

X =
```

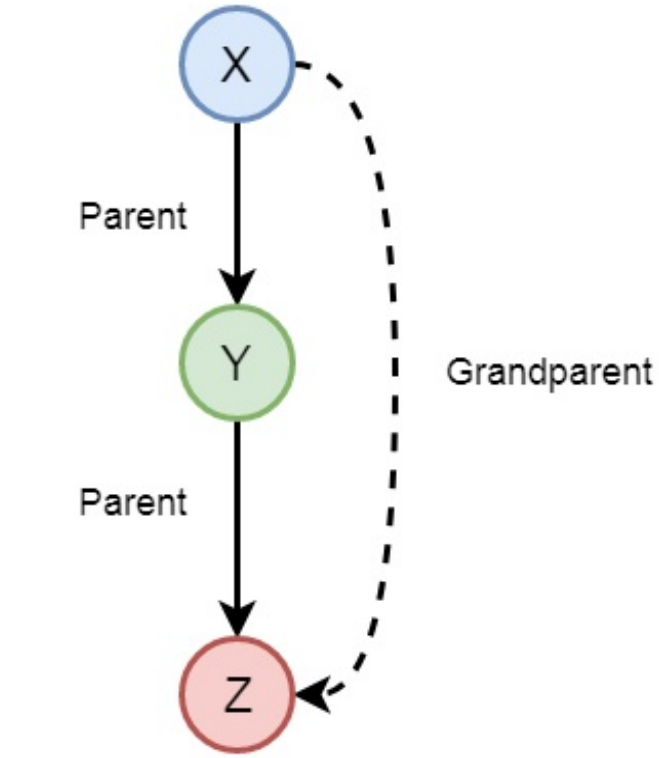
Prolog: Family Tree

- Output

```
| ?- sister(X,Y).

X = liz
Y = bob ? ;

X = ann
Y = pat ? ;

X = ann
Y = peter ? ;

X = pat
Y = ann ? ;

X = pat
Y = peter ? ;
```

Prolog: Family Tree

Now let us see some more relationships that we can make from the previous relationships of a family. So if we want to make a grandparent relationship, that can be formed as follows −



Grandparent Relationship

Prolog: Family Tree

```
grandparent(X,Y):-parent(X,Z),parent(Z,Y).
grandmother(X,Z):-mother(X,Y),parent(Y,Z).
grandfather(X,Z):-father(X,Y),parent(Y,Z).
wife(X,Y):-parent(X,Z),parent(Y,Z),female(X),male(Y).
uncle(X,Z):-brother(X,Y),parent(Y,Z).
| ?- uncle(X,Y).

X = peter
Y = jim ? ;

no
| ?- grandparent(X,Y).
```