Outline

- Representations And Mappings
- Approaches To Knowledge Representation
- Representation of Simple Facts In Logic
- Representing Instance And Isa Relationships
- Computable Functions and Predicates
- Resolution
- Procedural versus Declarative Knowledge
- Logic Programming
- Forward versus Backward Reasoning

Representation of Simple Facts In Logic

Introduction

- Logic
  - The logical formalism of a language is useful because it immediately suggests a powerful way of deriving new knowledge from old using mathematical deduction.
  - In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known.
- Proposition
  - A proposition is a statement, or a simple declarative sentence.
  - For example, “the book is expensive” is a proposition.
  - A proposition can be either true or false

Propositional logic

- Logical constants: true, false
- Propositional symbols: P, Q, S,... (atomic sentences)
- Propositions are combined by connectives:

$\wedge$	and [conjunction]
$\vee$	or [disjunction]
$\Rightarrow$	implies [implication]
$\neg$	not [negation]
$\forall$	For all
$\exists$	There exists

Propositional Logic

- Representing simple facts using propositional logic:
  1. It is raining

     RAINING
  2. It is sunny

     SUNNY
  3. It is windy

     WINDY
  4. If it is raining, then it is not sunny

     $\text{RAINING} \rightarrow \neg\, \text{SUNNY}$

Predicate Logic

- First-order Predicate logic (FOPL) is a formal language in which propositions are expressed in terms of predicates, variables and quantifiers.
- It is different from propositional logic which lacks quantifiers.
- It should be viewed as an extension to propositional logic, in which the notions of truth values, logical connectives, etc. still apply but propositional letters will be replaced by a newer notion of proposition involving predicates and quantifiers.
- A predicate is an expression of one or more variables defined on some specific domain.
- A predicate with variables can be made up of a proposition by either assigning a value to the variable or by quantifying the variable.

Well Formed Formula

- Well Formed Formula (wff) is a predicate holding any of the following –
  - All propositional constants and propositional variables are wffs
  - If x is a variable and Y is a wff, ∀Y and ∃x are also wff
  - Truth value and false values are wffs
  - Each atomic formula is a wff
  - All connectives connecting wffs are wffs

Facts Represented as Well Formed Formula in FOPL

1. Marcus was a man.

   $man(Marcus)$

2. Marcus was a Pompeian.

   $Pompeian(Marcus)$

3. All Pompeians were Romans.

   $\forall x: Pompeian(x) \rightarrow Roman(x)$

4. Caesar was a ruler.

   $ruler(Caesar)$

5. All Romans were either loyal to Caesar or hated him.

   $\forall x: Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

6. Everyone is loyal to someone.

   $\forall x: \exists x: loyalto(x,y)$

Facts Represented as Well Formed Formula in FOPL

7. People only try to assassinate rulers are not loyal to.

   $\forall x: \forall y: person(x) \land rular(y) \land tryassassinate(x,y) \rightarrow \neg loyalto(x,y)$

8. Marcus tried to assassinate Caesar.

   $tryassassinate(Marcus,Caesar)$

9. All men are people.

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

Use the above statements to answer the question: "Was Marcus loyal to Caesar?"

$\neg loyalto(Marcus,Caesar)$

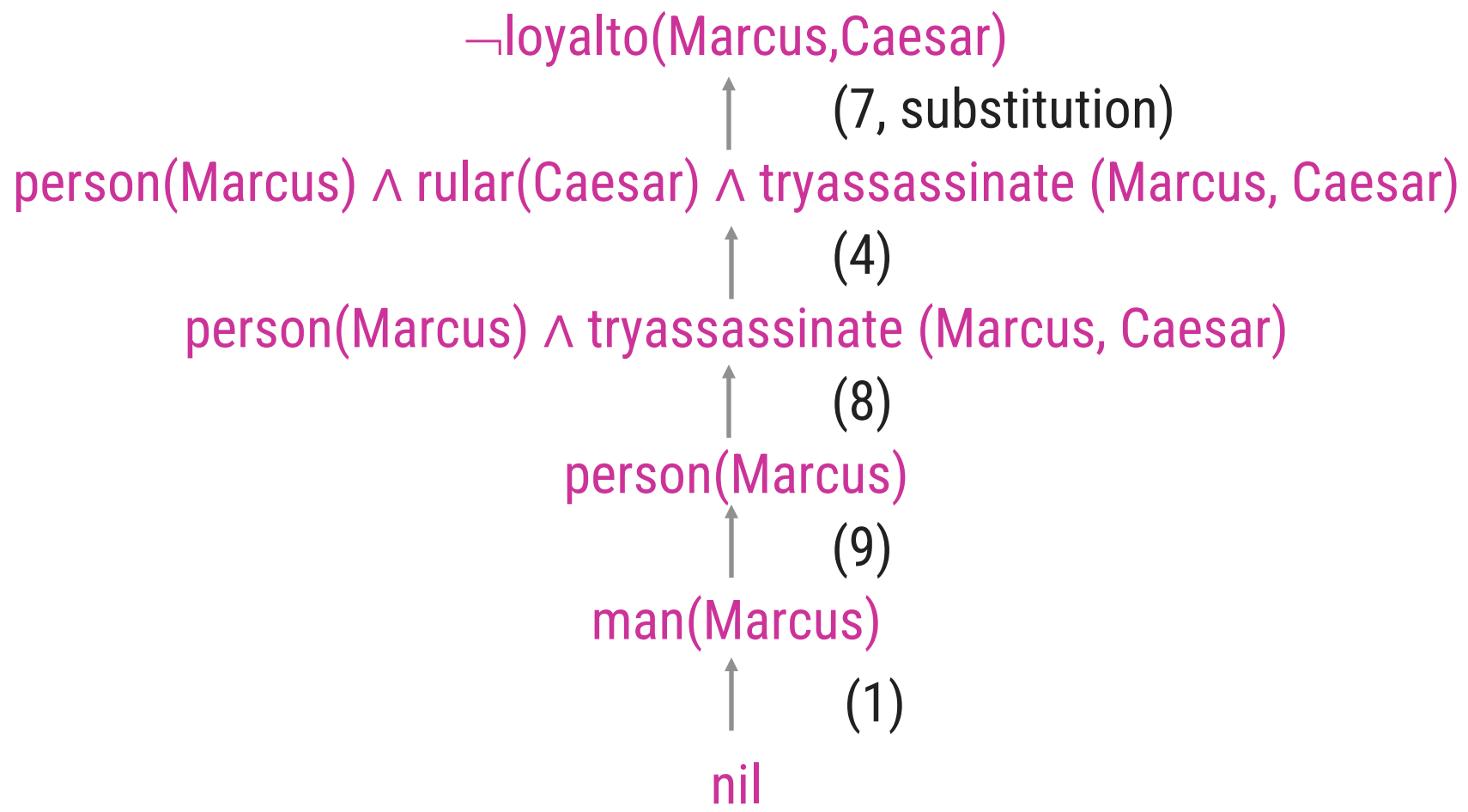
Use backward reasoning from desired goal:

Facts Represented as Well Formed Formula in FOPL

Use the above statements to answer the question: “Was Marcus loyal to Caesar?”

¬loyalto(Marcus,Caesar)

Use backward reasoning from the desired goal:



Representing INSTANCE and ISA Relationships

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

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

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

- Specific attributes instance and isa play important role particularly in a useful form of reasoning called property inheritance.
- The predicates instance and isa explicitly captured the relationships they are used to express, namely class membership and class inclusion.
- The predicate instance is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs.
- The use of the isa predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided.

Resolution

Introduction

- Resolution is a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form.
- Resolution produces proofs by refutation.
- In other words, to prove a statement (i.e., to show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (that it is un-satisfiable).
- The resolution procedure is a simple iterative process: at each step, two clauses, called the parent clauses, are compared (resolved), resulting into a new clause that has been inferred from them.
- The new clause represents ways that the two parent clauses interact with each other.

Conversion to Clause Form

- Algorithm: Conversion to Clause Form

1. Eliminatine $\rightarrow$, using the fact that $a \rightarrow b$ is equivalent to $\neg a \vee b$.
2. Reduce the scope of each $\neg$ to a single term, using the fact that
   - $\neg(\neg p) = p$,
   - DeMorga's laws, $\neg(a \wedge b) = \neg a \vee \neg b$, $\neg(a \vee b) = \neg a \wedge \neg b$
   - The standard correspondences between quantifiers [$\neg \forall x: P(x) = \exists x: \neg P(x)$ and $\neg \exists x: P(x) = \forall x: \neg P(x)$]
3. Standardize variables so that each quantifier binds a unique value.
   - For example, $\forall x: P(x) \vee \forall x: Q(x)$ would be converted to $\forall x: P(x) \vee \forall y: Q(y)$
4. Move all quantifiers to the left of the formula without changing their relative order.
5. Eliminate existential quantifiers.
6. Drop the prefix.
7. Convert the matrix into a conjunction of disjuncts. Try to exploit the associative property [av(bvc)=(a v b) v c] or distributive property [$(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$]
8. Create a separate clause corresponding to each conjunct.
9. Standardize apart the variables in each set of clauses generated in step 8.

Algorithm: Propositional Resolution

- ***Algorithm: Propositional Resolution***

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

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

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

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

The Unification Algorithm

- In propositional logic, it is easy to determine that two literals cannot both be true at the same time.
- Simply look for L and ¬L in predicate logic, this matching process is more complicated since the arguments of the predicates must be considered.
- For example, man(John) and ¬man(John) is a contradiction, while man(John) and ¬man(Spot) is not.
- Thus, to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical.
- There is a straightforward recursive procedure, called the unification algorithm, that does it.

The Unification Algorithm

- Algorithm: Unify(L1, 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 ith 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

- 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 obtained in 1.
3. Repeat until a contradiction is found, no progress can be made, or a predeterminedamount 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 an empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Facts Represented as Well Formed Formula in FOPL

1. Marcus was a man.

   $man(Marcus)$

2. Marcus was a Pompeian.

   $Pompeian(Marcus)$

3. All Pompeians were Romans.

   $\forall x: Pompeian(x) \rightarrow Roman(x)$

4. Caesar was a ruler.

   $ruler(Caesar)$

5. All Pompeians were either loyal to Caesar or hated him.

   $\forall x: Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

Facts Represented as Well Formed Formula in FOPL

6. Every one is loyal to someone.

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

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

   $\forall x: \forall y: \text{person}(x) \land \text{ruler}(y) \land \text{tryassassinate}(x, y) \rightarrow \neg \text{loyalto}(x, y)$

8. Marcus tried to assassinate Caesar.

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

Axioms in clause form

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).
6. $\forall x$: $\exists y$: loyalto(x, y).
7. $\forall x$: $\forall y$: person(x) $\wedge$ ruler(y) $\wedge$ tryassassinate(x, y) $\rightarrow$ $\neg$loyalto(x, y).
8. tryassassinate(Marcus, Caesar).

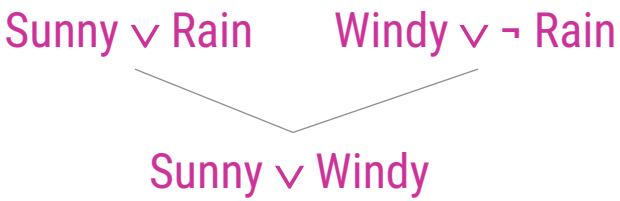
1. man(Marcus)
2. Pompeian(Marcus)
3. $\neg$Pompeian(x1) $\vee$ Roman(x1)
4. ruler(Caesar)
5. $\neg$Roman(x2) $\vee$ loyalto(x2, Caesar) $\vee$ hate(x2, Caesar)
6. loyalto(x3,fl(x3))
7. $\neg$person(x4) $\vee$ $\neg$ruler(y1) $\vee$ $\neg$tryassassinate(x4,y1) $\vee$ $\neg$ loyalto(x4,y1)
8. trayassassinate(Marcus,Caesar)

a $\rightarrow$ b is equivalent to $\neg$ a $\vee$ b

Prove: hate(Marcus, Caesar)

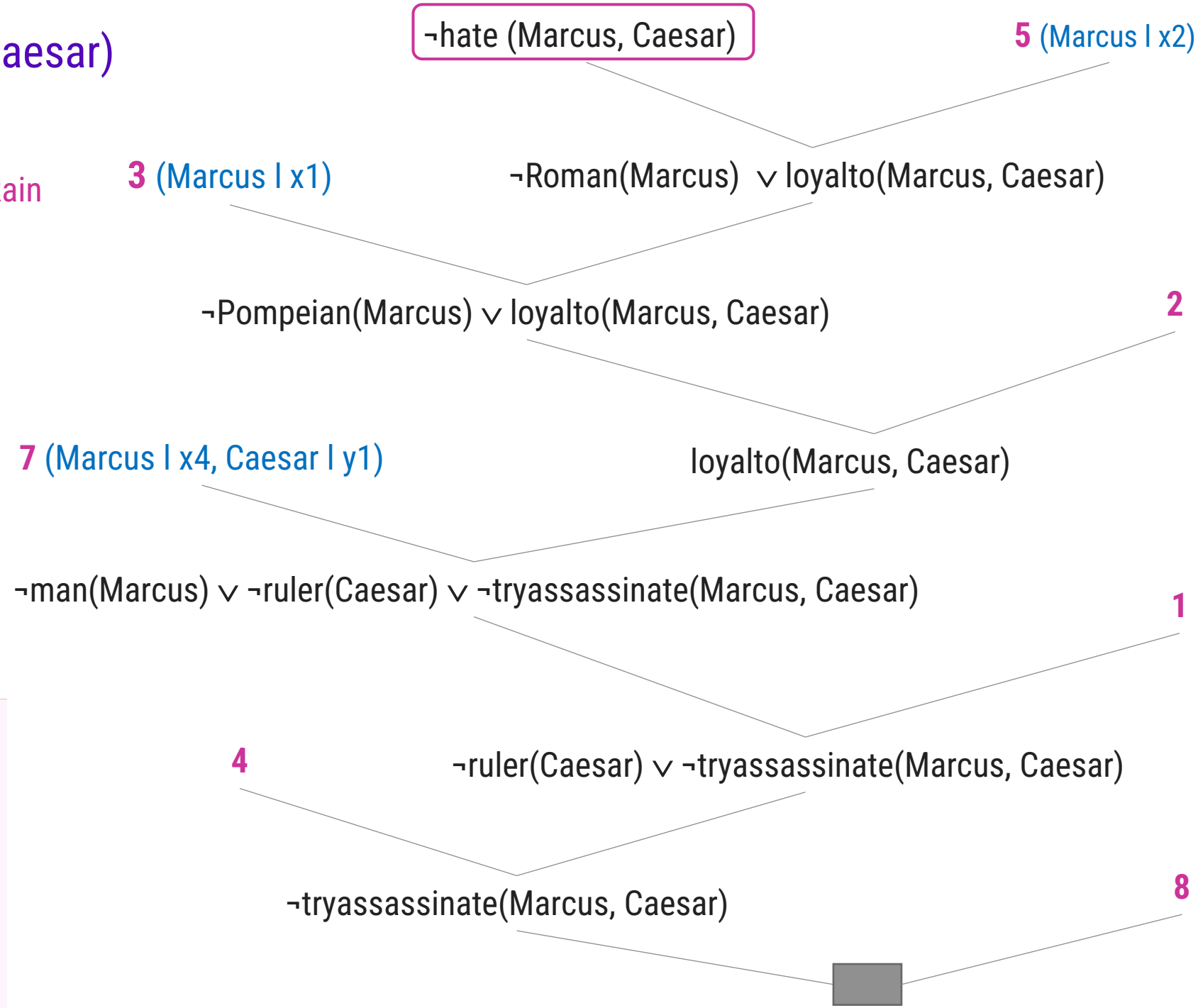
Resolution by Refutation

Prove: hate(Marcus, Caesar)



Final resolvent is an empty clause means that a contradiction is found in the initial assumption. So it is proved that hate (Marcus, Caesar).

1. man(Marcus)
2. Pompeian(Marcus)
3. ¬Pompeian(x1) ∨ Roman(x1)
4. ruler(Caesar)
5. ¬Roman(x2) ∨ loyalto(x2, Caesar) ∨ hate(x2, Caesar)
6. loyalto(x3,fl(x3))
7. ¬man(x4) ∨ ¬ruler(y1) ∨ ¬tryassassinate(x4,y1) ∨ ¬ loyalto(x4,y1)
8. trayassassinate(Marcus,Caesar)



Resolution by Refutation – Example

1. John likes all kinds of food.
2. Apples are food.
3. Chicken is food.
4. Anything anyone eats and isn’t killed by is food.
5. Bill eats peanuts and is still alive.
6. John eats everything Bill eats.

Resolution by Refutation – Example

Step 1 : Translate these sentences into formulas in FOPL

1. John likes all kinds of food.
   $\forall x \text{ Food}(x) \rightarrow \text{Likes}(\text{John}, x)$
2. Apples are food.
   $\text{Food}(\text{Apples})$
3. Chicken is food.
   $\text{Food}(\text{Chicken})$
4. Anything anyone eats and isn't killed by is food.
   $\forall x \exists y : \text{Eats}(y, x) \wedge \neg\text{KilledBy}(y, x) \rightarrow \text{Food}(x)$
5. Bill eats peanuts and is still alive.
   $\text{Eats}(\text{Bill}, \text{Peanuts}) \wedge \neg\text{KilledBy}(\text{Bill}, \text{Peanuts})$
6. John eats everything Bill eats.
   $\forall x : \text{Eats}(\text{Bill}, x) \rightarrow \text{Eats}(\text{John}, x)$

Axioms in clause form

Step 2 : Convert the formulas of step 1 into clause form.

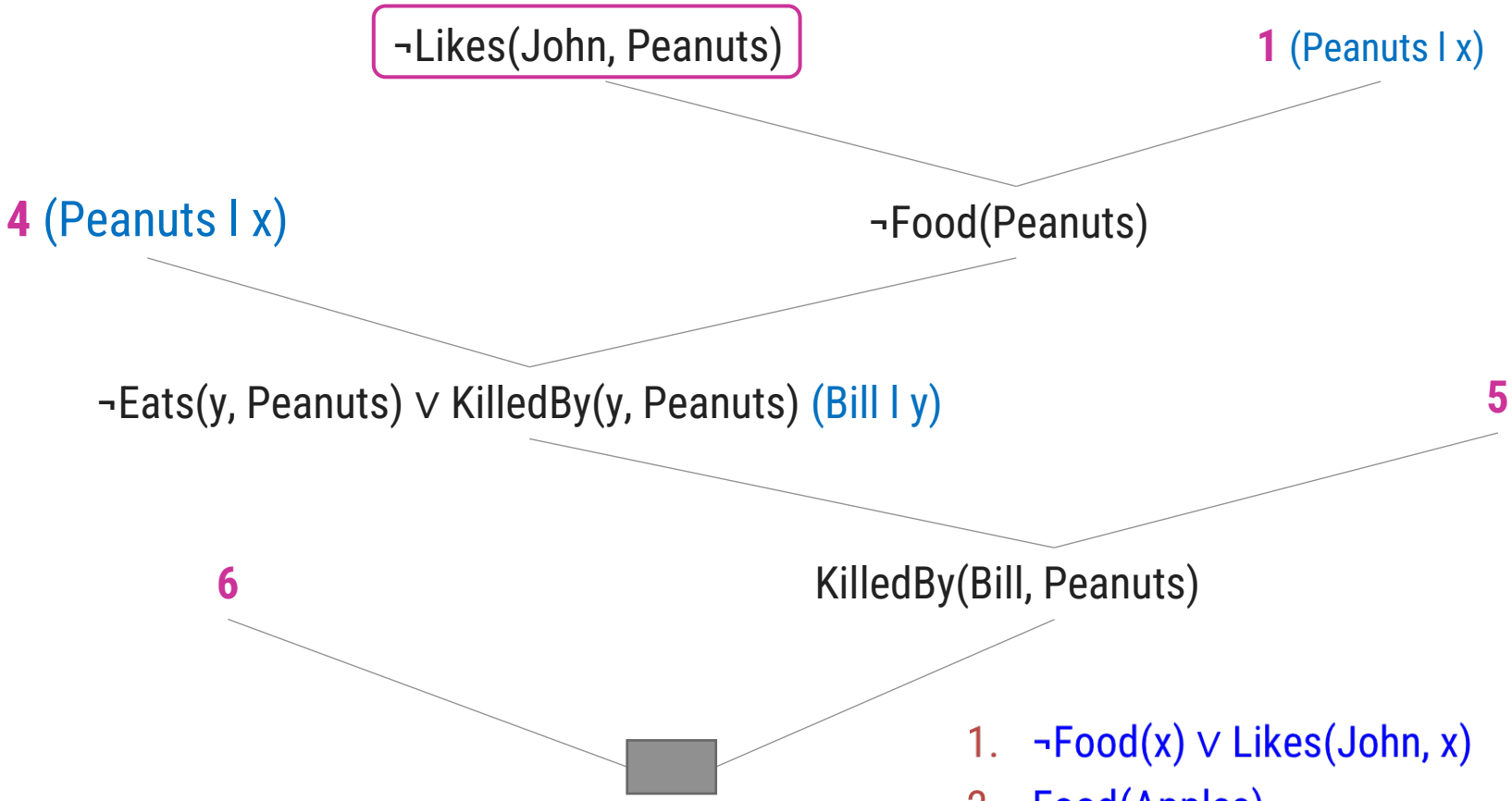
1. $\forall x$ Food(x) $\rightarrow$ Likes(John, x)
2. Food(Apples)
3. Food(Chicken)
4. $\forall x \exists y$: Eats(y, x) $\wedge \neg$KilledBy(y, x) $\rightarrow$ Food(x)
6. Eats(Bill, Peanuts) $\wedge \neg$KilledBy(Bill, Peanuts)
7. $\forall x$: Eats(Bill, x) $\rightarrow$ Eats(John, x)

1. $\neg$Food(x) $\vee$ Likes(John, x)
2. Food(Apples)
3. Food(Chicken)
4. $\neg$Eats(y, x) $\vee$ KilledBy(y, x) $\vee$ Food(x)
5. Eats(Bill, Peanuts)
6. $\neg$KilledBy(Bill, Peanuts)
7. $\neg$Eats(Bill, x) $\vee$ Eats(John, x)

Use resolution to prove that John likes peanuts.

Resolution by Refutation

Prove: Likes(John, Peanuts)



Final resolvent is an empty clause means that a contradiction is found in the initial assumption. So it is proved that

Likes(John, Peanuts)

1. ¬Food(x) ∨ Likes(John, x)
2. Food(Apples)
3. Food(Chicken)
4. ¬Eats(y, x) ∨ KilledBy(y, x) ∨ Food(x)
5. Eats(Bill, Peanuts)
6. ¬KilledBy(Bill, Peanuts)
7. ¬Eats(Bill, x) ∨ Eats(John, x)

Forward versus Backward Reasoning

- A search procedure must find a path between initial and goal states. There are two directions in which a search process could proceed.

Forward Reasoning	Backward Reasoning
Reason forward from the initial state	Reasoning backward from the goal states
1. It begins building a tree of moves by starting with the initial configuration at the root of the tree.	1. It begins building a tree of moves by starting with the goal node configuration at the root of the tree.
2. Generates the next level of tree by finding all rules whose left hand side matches against the root node. The right hand side is used to create new configurations.	2. Generate the next level of tree by finding all rules whose right hand side matches against the root node. The left hand side is used to create new configurations.
3. Forward Chaining searches for any conclusion. Suitable for planning, monitoring, control, and interpretation applications.	3. Backward chaining searches for only the required data. Suitable for diagnostic and debugging applications.

Thank You!