UNIT 3

LOGICAL REASONING

First Order Predicate Logic :syntax and symantics-Usage-Knowledge representation– Inference in First order logic:Unification – Forward Chaining- Backward Chaining – Resolution

3.1 First Order Logic

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation.

First-Order Logic is a logic which is sufficiently expressive to represent a good deal of our common sense knowledge.

- It is also either includes or forms the foundation of many other representation languages.
- It is also called as **First-Order Predicate calculus.**
- It is abbreviated as **FOL** or **FOPC**

FOL adopts the foundation of propositional logic with all its advantages to build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks.

The Syntax of natural language contains elements such as,

1. Nouns and noun phrases that refer to objects (Squares, pits, rumpuses)
2. Verbs and verb phrases that refer to among objects (is breezy, is adjacent to)

Some of these relations are functions-relations in which there is only one "Value" for a given "input". Whereas propositional logic assumes the world contains facts, first-order logic (like natural language) assumes the world contains

Objects: people, houses, numbers, colors, baseball games, wars, …

Relations: red, round, prime, brother of, bigger than, part of, comes between,…

Functions: father of, best friend, one more than, plus, …

3.2 SPECIFY THE SYNTAX AND SYMANTICS OF FIRST-ORDER LOGIC

3.2.1 Models for First-Order Logic

The domain of a model is DOMAIN the set of objects or domain elements it contains. The domain is required to be nonempty—every possible world must contain at least one object. Figure 8.2 shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown. The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation TUPLE is just the set of tuples of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set

{ ⟨Richard the Lionheart, King John⟩, ⟨ King John, Richard the Lionheart⟩ }

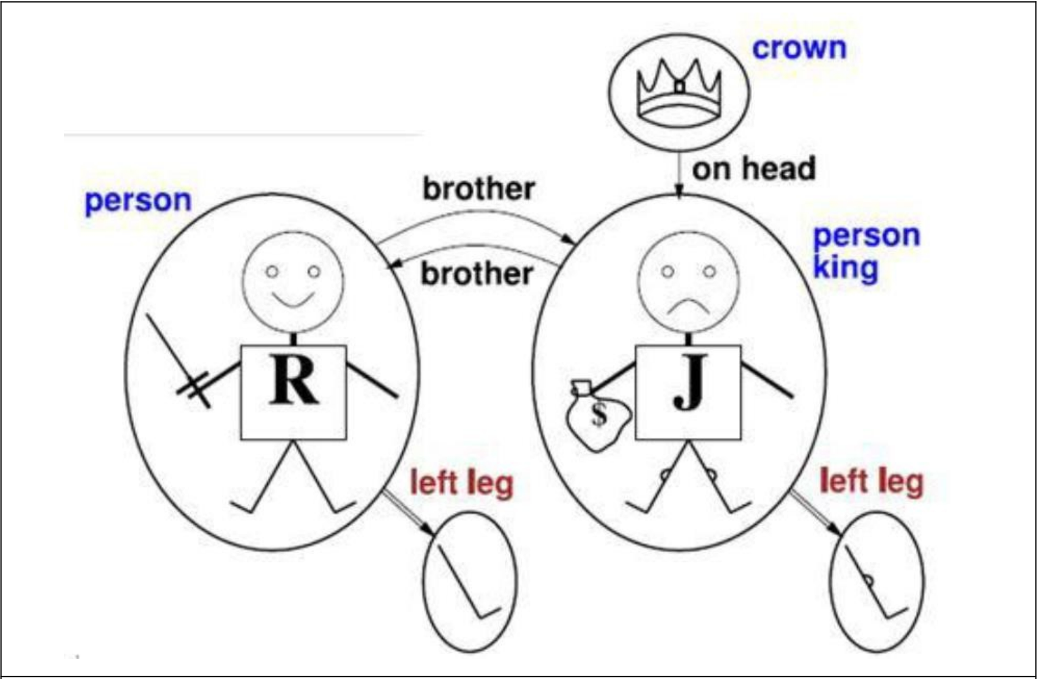


Figure 3.1 First-Order Logic in BNF Form

The crown is on King John's head, so the "on head" relation contains just one tuple, _ the crown, King John_. The "brother" and "on head" relations are binary relations — that is, they relate pairs of objects. Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:

⟨Richard the Lionheart⟩ → Richard's left leg
⟨King John⟩ → John's left leg

The five objects are,

- Richard the Lionheart
- His younger brother
- The evil King John
- The left legs of Richard and John
- A crown

- The objects in the model may be related in various ways, In the figure Richard and John are brothers.
- Formally speaking, a relation is just the set of tuples of objects that are related.
- A tuple is a collection of Objects arranged in a fixed order and is written with angle brackets surrounding the objects.
- Thus, the brotherhood relation in this model is the set **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**
- The crown is on King John's head, so the "on head" relation contains just one tuple, (the crown, King John).
    - The relation can be binary relation relating pairs of objects (Ex:- "Brother") or unary relation representing a common object (Ex:- "Person" representing both Richard and John)

Certain kinds of relationships are best considered as functions that relates an object to exactly one object.

- For Example:- each person has one left leg, so the model has a unary "left leg" function that includes the following mappings (Richard the Lionheart)-----> Richard's left leg (King John)---- > John's left leg

3.2.2 Symbols and Interpretations:

The basic syntactic elements of first-order logic are the symbols that stand for **objects, relations** and **functions**

Kinds of Symbols

The symbols come in three kinds namely,

1)Constant Symbols standing for **Objects** (Ex:- Richard)
2)Predicate Symbols standing for **Relations** (Ex:- King)
3)Function Symbols stands for **functions** (Ex:-Left Leg)

o Symbols will begin with uppercase letters
o The choice of names is entirely up to the user
o Each predicate and function symbol comes with an arity
o Arity fixes the number of arguments.

- The semantics must relate sentences to models in order to determine truth.
- To do this, an interpretation is needed specifying exactly which **objects, relations** and **functions** are referred to by the **constant, predicate and function symbols**.
- One possible interpretation called as the intended interpretation- is as follows;
- **Richard** refers to **Richard the Lion heart** and **John** refers to the **evil King John.**
- **Brother** refers to the brotherhood relation, that is the set of tuples of objects given in equation **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**
- **On Head** refers to the “on head” relation that holds between the crown and King John; **Person, King** and **Crown** refer to the set of objects that are persons, kings and crowns.
- **Left leg** refers to the “left leg” function, that is, the mapping given in **{(Richard the Lion heart, King John), (King John, Richard the Lionheart)}**

A complete description from the formal grammar is as follows

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \\
 & | & (\ \textit{Sentence Connective Sentence}\ ) \\
 & | & \textit{Quantifier Variable},\ldots\ \textit{Sentence} \\
 & | & \neg\ \textit{Sentence} \\
 & & \\
\textit{AtomicSentence} & \rightarrow & \textit{Predicate}(\textit{Term},\ldots)\ |\ \textit{Term} = \textit{Term} \\
 & & \\
\textit{Term} & \rightarrow & \textit{Function}(\textit{Term},\ldots) \\
 & | & \textit{Constant} \\
 & | & \textit{Variable} \\
 & & \\
\textit{Connective} & \rightarrow & \Rightarrow\ |\ \wedge\ |\ \vee\ |\ \Leftrightarrow \\
\textit{Quantifier} & \rightarrow & \forall\ |\ \exists \\
\textit{Constant} & \rightarrow & A\ |\ X_1\ |\ \textit{John}\ |\ \ldots \\
\textit{Variable} & \rightarrow & a\ |\ x\ |\ s\ |\ \ldots \\
\textit{Predicate} & \rightarrow & \textit{Before}\ |\ \textit{HasColor}\ |\ \textit{Raining}\ |\ \ldots \\
\textit{Function} & \rightarrow & \textit{Mother}\ |\ \textit{LeftLeg}\ |\ \ldots
\end{array}
$$

3.2.3 Terms

A term is a logical expression that refers TERM to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use Left Leg (John). The formal semantics of terms is straightforward. Consider a term f(t1, . . . , tn). The function symbol f refers to some function in the model.

3.2.4 Atomic sentences

Atomic sentence (or atom for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as Brother (Richard, John). Atomic sentences can have complex terms as arguments. Thus, Married (Father (Richard),Mother (John)) states that Richard the Lionheart's father is married to King John's mother.

3.2.5 Complex Sentences

We can use logical connectives to construct more complex sentences, with the same syntax and semantics as in propositional calculus

¬Brother (LeftLeg (Richard), John)

Brother (Richard, John) ∧ Brother (John, Richard)

King(Richard) ∨ King(John)

¬King(Richard) ⇒ King(John).

3.2.6 Quantifiers

Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name. First-order logic contains two standard quantifiers, called universal and existential

Universal quantification (∀)

"All kings are persons," is written in first-order logic as

∀x King(x) ⇒ Person(x)

∀ is usually pronounced "For all. . ." Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called a variable. A term with no variables is called a **ground term.**Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

x → Richard the Lionheart,

x → King John, x → Richard's left leg,

x → John's left leg,

x → the crown.

The universally quantified sentence ∀ x King(x) ⇒ Person(x) is true in the original model if the sentence King(x) ⇒ Person(x) is true under each of the five extended interpretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

Richard the Lionheart is a king ⇒ Richard the Lionheart is a person.

King John is a king ⇒ King John is a person.

Richard's left leg is a king ⇒ Richard's left leg is a person.

John's left leg is a king ⇒ John's left leg is a person.

The crown is a king ⇒ the crown is a person.

Existential quantification (∃)

Universal quantification makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write

∃x Crown(x) ∧OnHead(x, John)

∃x is pronounced "There exists an x such that . . ." or "For some x . . ." More precisely, ∃x P is true in a given model if P is true in at least one extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

Richard the Lionheart is a crown ∧ Richard the Lionheart is on John's head;

King John is a crown ∧ King John is on John's head;

Richard's left leg is a crown ∧ Richard's left leg is on John's head;

John's left leg is a crown ∧ John's left leg is on John's head;

The crown is a crown ∧ the crown is on John's head.

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Just as ⇒ appears to be the natural connective to use with ∀, ∧ is the natural connective to use with ∃.

Using ∧ as the main connective with ∀ led to an overly strong statement in the example in the previous section; using ⇒ with ∃ usually leads to a very weak statement, indeed. Consider the following sentence:

∃x Crown(x) ⇒OnHead(x, John)

Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

Richard the Lionheart is a crown ⇒ Richard the Lionheart is on John's head;

King John is a crown ⇒ King John is on John's head;

Richard's left leg is a crown ⇒ Richard's left leg is on John's head;

and so on. Now an implication is true if both premise and conclusion are true, or if its premise is false. So if Richard the Lionheart is not a crown, then the first assertion is true and the existential is satisfied. So, an existentially quantified implication sentence is true whenever any object fails to satisfy the premise

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "Brothers are siblings" can be written as

∀x∀ ***y Brother (x, y) ⇒ Sibling(x, y)***

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

∀***x, y Sibling(x, y)*** ⇔ ***Sibling(y, x)***. In other cases we will have mixtures. "Everybody loves somebody" means that for every person, there is someone that person loves:

∀x∃ ***y Loves(x, y).***

On the other hand, to say "There is someone who is loved by everyone," we write

∃y∀ ***x Loves(x, y).***

The order of quantification is therefore very important. It becomes clearer if we insert parentheses.

∀***x (***∃ ***y Loves(x, y))*** says that everyone has a particular property, namely, the property that they love someone. On the other hand,

∃***y (***∀ ***x Loves(x, y))*** says that someone in the world has a particular property, namely the property of being loved by everybody.

Connections between ∀ ***and*** ∃

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

∀x¬Likes(x, Parsnips) is equivalent to ¬

∀x¬Likes(x, Parsnips) is equivalent to ¬∃ x Likes(x, Parsnips). We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream:

∀x Likes(x, IceCream) is equivalent to ¬∃ x ¬Likes(x, IceCream).

3.2.7 Equality

We can use the equality symbol to signify that two terms refer to the same object. For example,

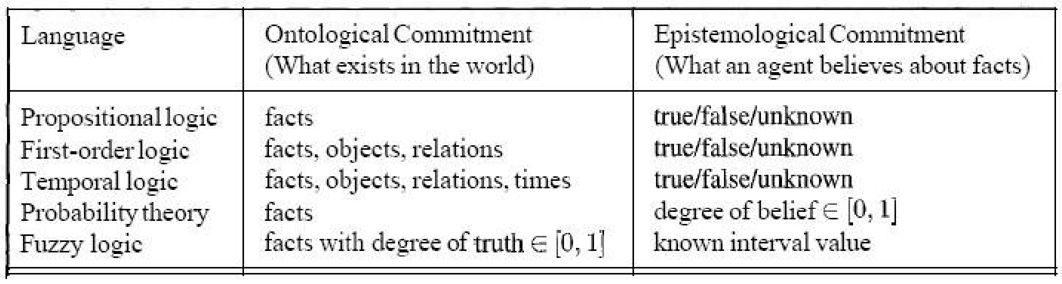
Father (John)=Henry

says that the object referred to by Father (John) and the object referred to by Henry are the same.

The equality symbol can be used to state facts about a given function, as we just did for the Father symbol. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, we would write

∃x, y Brother (x,Richard) ∧ Brother (y,Richard) ∧¬(x=y).

Compare different knowledge representation languages



Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure 3.2 Formal languages and their ontological and epistemological commitments

What are the syntactic elements of First Order Logic?

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, come in three kinds:

a) constant symbols, which stand for objects;
b) predicate symbols, which stand for relations;
c) and function symbols, which stand for functions.

We adopt the convention that these symbols will begin with uppercase letters. Example:

Constant symbols :
Richard and John;
predicate symbols :
Brother, On Head, Person, King, and Crown; function symbol : LeftLeg.

Quantifiers

Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name if a logic that allows object is found.

It has two type,
The following are the types of standard quantifiers,
- Universal
- Existential

Universal quantification

Explain Universal Quantifiers with an example.

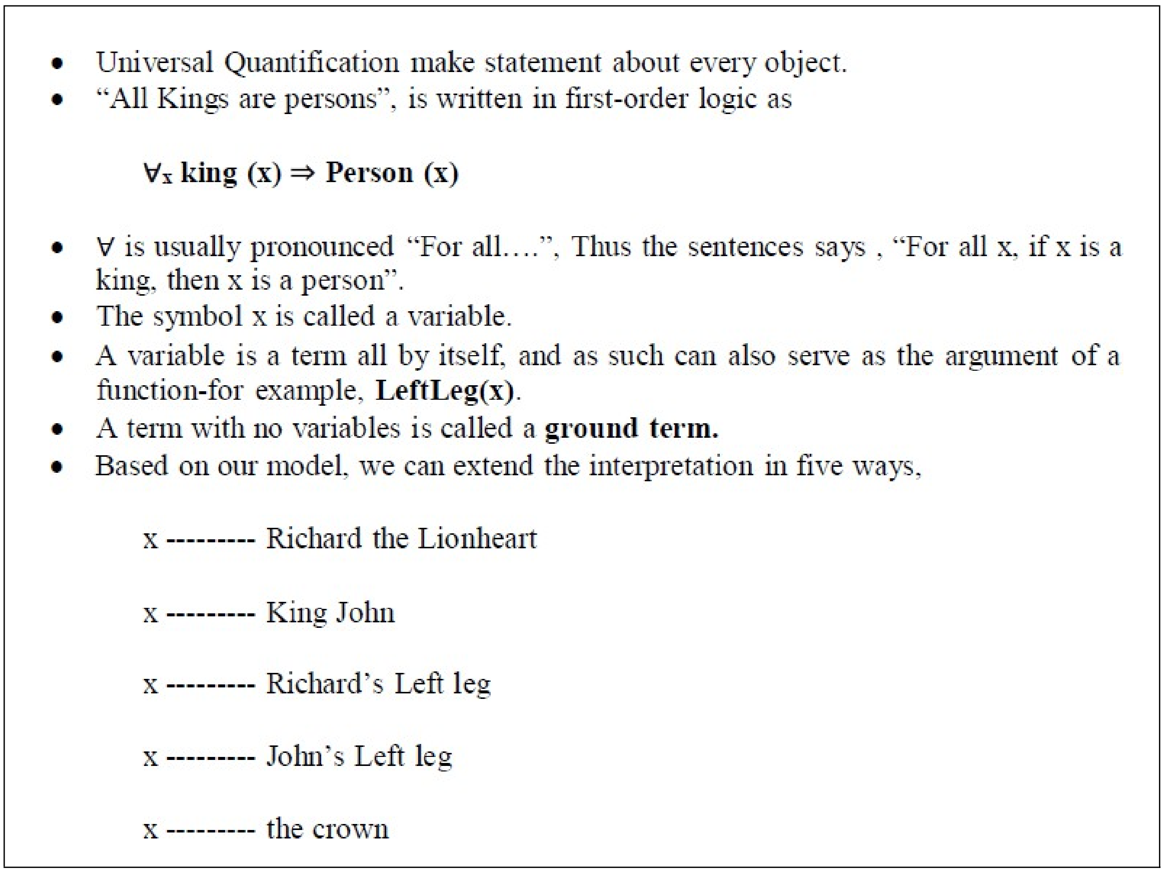
Rules such as "All kings are persons," is written in first-order logic as

$$\forall x\ King(x) \Rightarrow Person(x)$$

where $\forall$ is pronounced as " For all.."

Thus, the sentence says, "For all *x,* if *x* is a king, then is a person." The symbol x is called a variable(lower case letters)

The sentence $\forall$**x** P, where P is a logical expression says that P is true for every object x.



- Universal Quantification make statement about every object.
- "All Kings are persons", is written in first-order logic as

  $\forall_x$ **king (x) $\Rightarrow$ Person (x)**

- $\forall$ is usually pronounced "For all….", Thus the sentences says , "For all x, if x is a king, then x is a person".
- The symbol x is called a variable.
- A variable is a term all by itself, and as such can also serve as the argument of a function-for example, **LeftLeg(x)**.
- A term with no variables is called a **ground term.**
- Based on our model, we can extend the interpretation in five ways,

  x --------- Richard the Lionheart

  x --------- King John

  x --------- Richard's Left leg

  x --------- John's Left leg

  x --------- the crown

The universally quantified sentence is equivalent to asserting the following five sentences

Richard the Lionheart------- Richard the Lionheart is a person
King John is a King------- King John is a Person
Richard’s left leg is King--------- Richard’s left leg is a person
John’s left leg is a King--------- John’s left leg is a person
The crown is a King--------- The crown is a Person

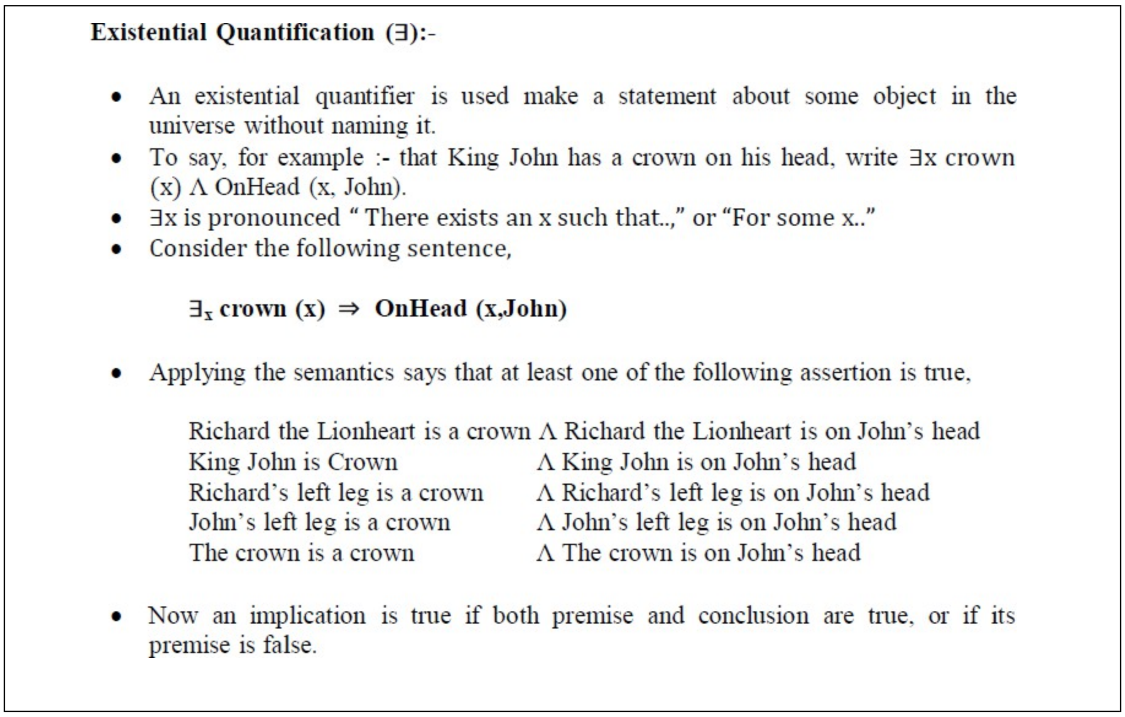
Existential quantification

Universal quantification makes statements about every object.

It is possible to make a statement about some object in the universe without naming it,by using an existential quantifier.

Example

“King John has a crown on his head”
□x Crown(x) ^ OnHead(x,John)
□x is pronounced “There exists an x such that..” or “ For some x ..”



Existential Quantification (∃):-

- An existential quantifier is used make a statement about some object in the universe without naming it.
- To say, for example :- that King John has a crown on his head, write ∃x crown (x) Λ OnHead (x, John).
- ∃x is pronounced “ There exists an x such that..,” or “For some x..”
- Consider the following sentence,

  $\exists_x$ **crown (x) ⇒ OnHead (x,John)**

- Applying the semantics says that at least one of the following assertion is true,

  Richard the Lionheart is a crown Λ Richard the Lionheart is on John’s head
  King John is Crown Λ King John is on John’s head
  Richard’s left leg is a crown Λ Richard’s left leg is on John’s head
  John’s left leg is a crown Λ John’s left leg is on John’s head
  The crown is a crown Λ The crown is on John’s head

- Now an implication is true if both premise and conclusion are true, or if its premise is false.

Nested Quantifiers

More complex sentences are expressed using multiple quantifiers.
The following are the some cases of multiple quantifiers,
The simplest case where the quantifiers are of the same type.

For Example:- “Brothers are Siblings” can be written as

$$\forall_x \forall_{y,} \mathbf{Brother\ (x,y) \Rightarrow sibling\ (x,y)}$$

Consecutive quantifiers of the same type can be written as one quantifier with

several variables. For Example:- to say that siblinghood is a symmetric relationship as

$$\forall_{x,y} \mathbf{sibling\ (x,y) \Leftrightarrow sibling\ (y,z)}$$

3.3 Using First Order Logic

We have seen some simple sentences illustrating the various aspects of logical syntax; in this section, we provide more systematic representations of some simple **domains**. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

3.3.1 Assertions and queries in first-order logic

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions**. For example, we can assert that John is a king, Richard is a person, and all kings are persons:

TELL(*KB, King*(*John*)) .
TELL(*KB, Person*(*Richard*)) .
TELL(*KB,* 6 *x King*(*x*) ⇒ *Person*(*x*)) .

We can ask questions of the knowledge base using ASK. For example,

ASK(*KB, King*(*John*))

returns *true*. Questions asked with ASK are called **queries** or **goals**. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the two preceding assertions, the query

ASK(*KB, Person*(*John*))

should also return *true*. We can ask quantified queries, such as ASK(*KB,* E *x*

Person(*x*)) .

The answer is *true*, but this is perhaps not as helpful as we would like. It is rather like answering “Can you tell me the time?” with “Yes.” If we want to know what value of *x* makes the sentence true, we will need a different function, ASKVARS, which we call with

ASKVARS(*KB, Person*(*x*))

and which yields a stream of answers. In this case there will be two answers: {*x/John*} and {*x/Richard*}. Such an answer is called a **substitution** or **binding list**.ASKVARS is usually reserved for knowledge bases consisting solely of Horn clauses, because in such knowledge bases every way of making the query true will bind the variables to specific values. That is not the case with first-order logic; if *KB* has been told *King*(*John*) V *King*(*Richard*), then there is no binding to *x* for the query E *x King*(*x*), even though the query is true.

8.3.2 THE WUMPUS WORLD

The wumpus world is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence. A sample wumpus world is shown in Figure

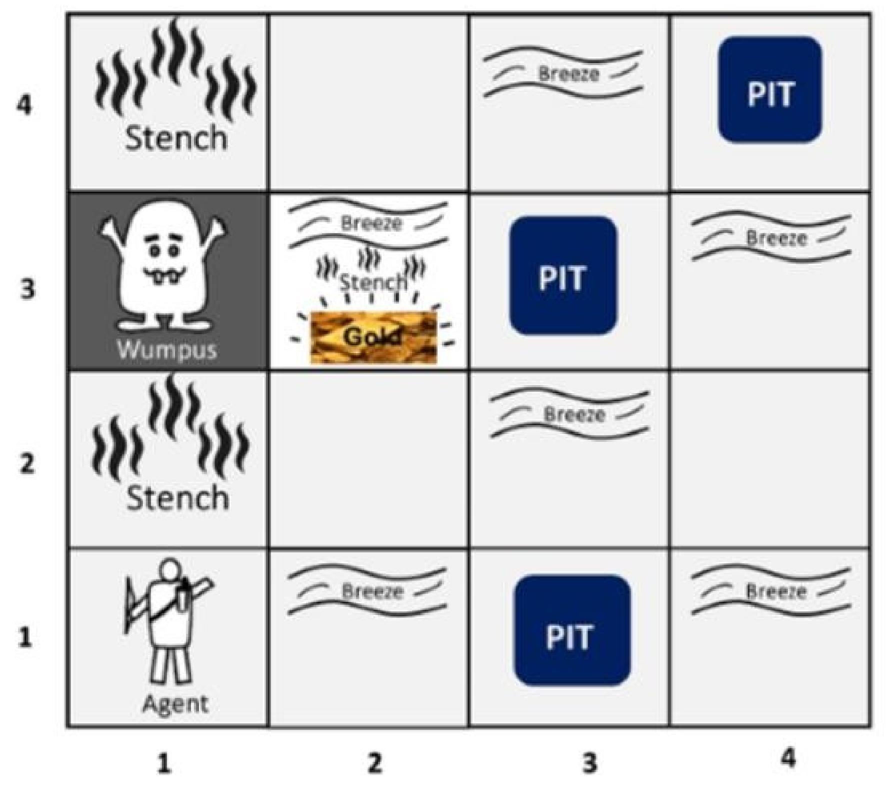


Figure 3.3 wumpus world

To specify the agent's task, we specify its percepts, actions, and goals. In the wumpus world, these are as follows:

- In the square containing the wumpus and in the directly (not diagonally) adjacent squares the agent will perceive a stench.
- In the squares directly adjacent to a pit, the agent will perceive a breeze.
- In the square where the gold is, the agent will perceive a glitter.
- When an agent walks into a wall, it will perceive a bump.
- When the wumpus is killed, it gives out a woeful scream that can be perceived anywhere in the cave.
- The percepts will be given to the agent in the form of a list of five symbols; for example, if there is a stench, a breeze, and a glitter but no bump and no scream, the agent will receive the percept [Stench, Breeze, Glitter, None, None]. The agent cannot perceive its own location.
- Just as in the vacuum world, there are actions to go forward, turn right by 90°, and turn left by 90°. In addition, the action Grab can be used to pick up an object that is in the same square as the agent. The action Shoot can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits and kills the wumpus or hits the wall. The agent only has one arrow, so only the first Shoot action has any effect.

The wumpus agent receives a percept vector with five elements. The corresponding first order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what. We use integers for time steps. A typical percept sentence would be

Percept ([Stench, Breeze, Glitter, None, None], 5).

Here, Percept is a binary predicate, and Stench and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

Turn(Right), Turn(Left), Forward, Shoot, Grab, Climb.

To determine which is best, the agent program executes the query

ASKVARS(∃ a BestAction(a, 5)),

which returns a binding list such as {a/Grab}. The agent program can then return Grab as the action to take. The raw percept data implies certain facts about the current state.

For example:

$\forall$***t,s,g,w,c Percept ([s,Breeze,g,w,c],t) ⇒ Breeze(t),***

$\forall$***t,s,b,w,c*** Percept ***([s,b,Glitter,w,c],t) ⇒ Glitter (t)***

These rules exhibit a trivial form of the reasoning process called perception. Simple "reflex" behavior can also be implemented by quantified implication sentences.

For example, we have ∀ ***t Glitter (t) ⇒BestAction(Grab, t)***.

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion

BestAction***(Grab, 5)***—that is, Grab is the right thing to do. For example, if the agent is at a square and perceives a breeze, then that square is breezy:

∀***s, t At(Agent, s, t)*** ∧ ***Breeze(t)*** ⇒ ***Breezy(s).***

It is useful to know that a square is breezy because we know that the pits cannot move about. Notice that Breezy has no time argument. Having discovered which places are breezy (or smelly) and, very important, not breezy (or not smelly), the agent can deduce where the pits are (and where the wumpus is). first-order logic just needs one axiom:

∀***s Breezy(s)*** ⇔∃***r Adjacent (r, s)*** ∧ ***Pit(r).***

3.4 Knowledge Engineering in First Order Logic

The general process of knowledge-base construction— a process called knowledge engineering. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain.

3.4.1 The Knowledge-Engineering Process

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1)Identify the task. The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. The task will determine what knowledge must be represented in order to connect problem instances to answers.

2) Assemble the relevant knowledge. The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called **knowledge acquisition**. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

3) Decide on a vocabulary of predicates, functions, and constants. That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering *style.* Like programming style, this can have a significant impact on the eventual success of the project.

4) Encode general knowledge about the domain. The knowledge engineer writes down the axioms for all the vocabulary *terms*. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

5) *Encode a description of the specific problem instance.* If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about in- stances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is sup- plied with additional sentences in the same way that traditional programs are supplied with input data.
6) *Pose queries to the inference procedure and get answers.* This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.
7) *Debug the knowledge base.* Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct *for the knowledge base as written*, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting.

3.5 First Order Logic Inference

3.5.1 Unification

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called unification and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a unifier for them if one exists: ***UNIFY(p, q)=θ*** where ***SUBST(θ, p)= SUBST(θ, q)***.

Suppose we have a query ***AskVars(Knows(John, x)):*** whom does John know? Answers can be found by finding all sentences in the knowledge base that unify with ***Knows(John, x).***

Here are the results of unification with four different sentences that might be in the knowledge base: ***UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane}***

UNIFY(Knows(John, x), Knows(y, Bill)) = {x/Bill, y/John}

UNIFY(Knows(John, x), Knows(y, Mother (y))) = {y/John, x/Mother (John)}
UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail.

The last unification fails because x cannot take on the values John and Elizabeth at the same time. Now, remember that ***Knows(x, Elizabeth)*** means "Everyone knows Elizabeth," so we should be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by standardizing apart one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in

Knows(x, Elizabeth) to x17 (a new variable name) without changing its meaning.

Now the unification will work

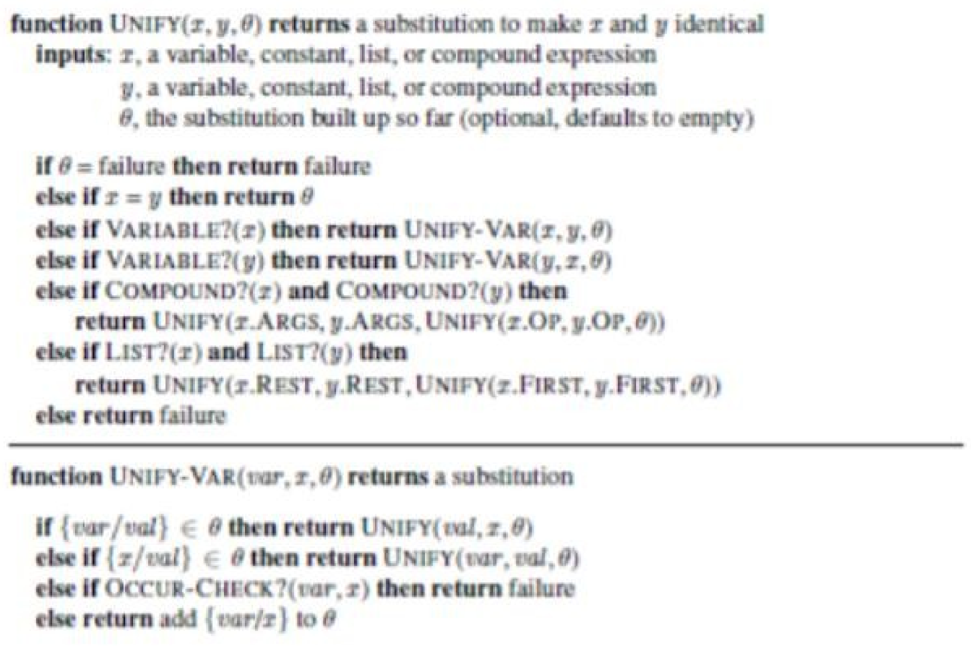
UNIFY(Knows(John, x), Knows(x17, Elizabeth)) = {x/Elizabeth, x17/John} UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier.

For example,

UNIFY(Knows(John, x), Knows(y, z)) could return
{y/John, x/z} or {y/John, x/John, z/John}.

The first unifier gives ***Knows(John, z)*** as the result of unification, whereas the second gives ***Knows(John, John).*** The second result could be obtained from the first by an additional substitution ***{z/John};*** we say that the first unifier is more general than the second, because it places fewer restrictions on the values of the variables. An algorithm for computing most general unifiers is shown in Figure.

The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed.



```
function UNIFY($x, y, \theta$) returns a substitution to make $x$ and $y$ identical
  inputs: $x$, a variable, constant, list, or compound expression
          $y$, a variable, constant, list, or compound expression
          $\theta$, the substitution built up so far (optional, defaults to empty)

  if $\theta$ = failure then return failure
  else if $x = y$ then return $\theta$
  else if VARIABLE?($x$) then return UNIFY-VAR($x, y, \theta$)
  else if VARIABLE?($y$) then return UNIFY-VAR($y, x, \theta$)
  else if COMPOUND?($x$) and COMPOUND?($y$) then
      return UNIFY($x$.ARGS, $y$.ARGS, UNIFY($x$.OP, $y$.OP, $\theta$))
  else if LIST?($x$) and LIST?($y$) then
      return UNIFY($x$.REST, $y$.REST, UNIFY($x$.FIRST, $y$.FIRST, $\theta$))
  else return failure
```

```
function UNIFY-VAR($var, x, \theta$) returns a substitution

  if $\{var/val\} \in \theta$ then return UNIFY($val, x, \theta$)
  else if $\{x/val\} \in \theta$ then return UNIFY($var, val, \theta$)
  else if OCCUR-CHECK?($var, x$) then return failure
  else return add $\{var/x\}$ to $\theta$
```

Figure 3.4 Recursively explore

In artificial intelligence, forward and backward chaining is one of the important topics, but before understanding forward and backward chaining lets first understand that from where these two terms came.

Inference engine

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts. The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:

a. Forward chaining
b. Backward chaining

Horn Clause and Definite clause

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the first-order definite clause.nt

Definite clause: A clause which is a disjunction of literals with exactly one positive literal is known as a definite clause or strict horn clause.

Horn clause: A clause which is a disjunction of literals with at most one positive literal is known as horn clause. Hence all the definite clauses are horn clauses.

Example: $(\neg p \vee \neg q \vee k)$. It has only one positive literal k.
It is equivalent to $p \wedge q \rightarrow k$.

3.5.2Forward Chaining

Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.

The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

Properties of Forward-Chaining

- It is a down-up approach, as it moves from bottom to top.
- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.

- Forward-chaining approach is also called as data-driven as we reach to the goal using available data.
- Forward -chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Consider the following famous example which we will use in both approaches:

Example

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen."

Prove that "Robert is criminal."

To solve the above problem, first, we will convert all the above facts into first-order definite clauses, and then we will use a forward-chaining algorithm to reach the goal.

Facts Conversion into FOL

- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
  American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) …(1)
- Country A has some missiles. ∃p Owns(A, p) ∧ Missile(p). It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.
  Owns(A, T1) …(2)
  Missile(T1) …(3)
- All of the missiles were sold to country A by Robert.
  ∀p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) …(4)
- Missiles are weapons.
  Missile(p) → Weapons (p) …(5)
- Enemy of America is known as hostile.
  Enemy(p, America) →Hostile(p) …(6)
- Country A is an enemy of America.
  Enemy (A, America) …(7)
- Robert is American
  American(Robert). …(8)

Forward chaining proof

Step-1

In the first step we will start with the known facts and will choose the sentences which do not have implications, such as: American (Robert), Enemy(A, America), Owns(A, T1), and Missile(T1). All these facts will be represented as below.

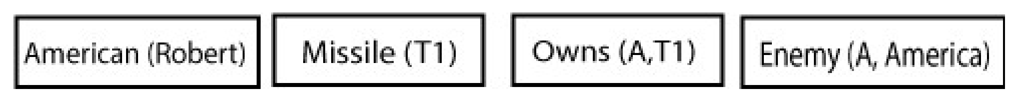


Figure 3.5

Step-2

At the second step, we will see those facts which infer from available facts and with satisfied premises.

Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

Rule-(2) and (3) are already added.

Rule-(4) satisfy with the substitution {p/T1}, so Sells (Robert, T1, A) is added, which infers from the conjunction of Rule (2) and (3).

Rule-(6) is satisfied with the substitution(p/A), so Hostile(A) is added and which infers from Rule-(7).

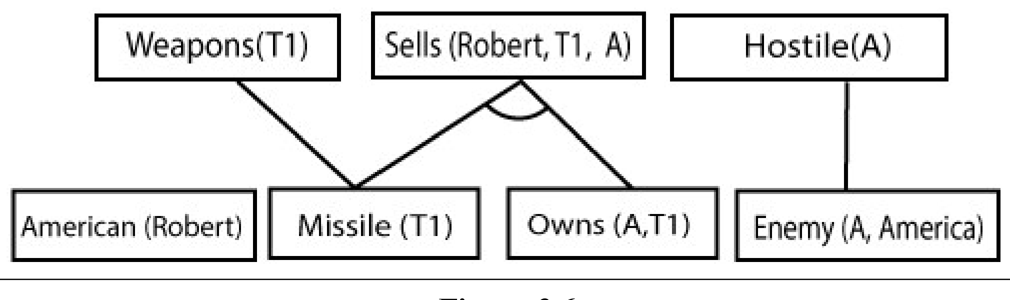


Figure 3.6

Step-3

At step-3, as we can check Rule-(1) is satisfied with the substitution {p/Robert, q/T1, r/A}, so we can add Criminal (Robert) which infers all the available facts. And hence we reached our goal statement.

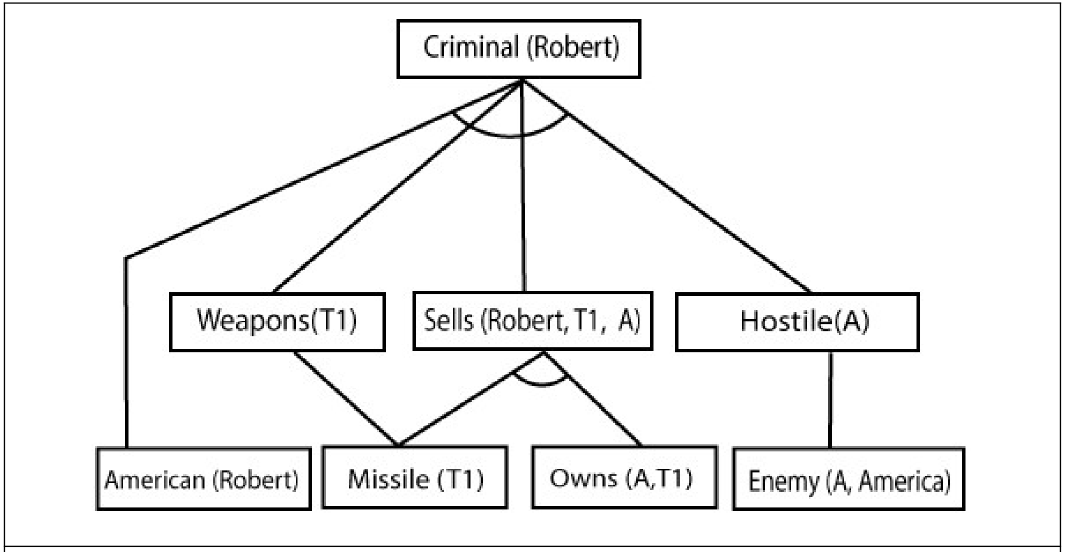


Figure 3.7

Hence it is proved that Robert is Criminal using forward chaining approach.

3.5.3 Backward Chaining

Backward-chaining is also known as a backward deduction or backward reasoning method when using an inference engine. A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

Properties of backward chaining

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a depth-first search strategy for proof.

Example

In backward-chaining, we will use the same above example, and will rewrite all the rules.

- American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) …(1)
  Owns(A, T1) …(2)

- Missile(T1)
- ?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) …(4)
- Missile(p) → Weapons (p) …(5)
- Enemy(p, America) →Hostile(p) …(6)
- Enemy (A, America) …(7)
- American (Robert). …(8)

Backward-Chaining proof

In Backward chaining, we will start with our goal predicate, which is Criminal (Robert), and then infer further rules.

Step-1

At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.

Step-2

At the second step, we will infer other facts form goal fact which satisfies the rules. So as we can see in Rule-1, the goal predicate Criminal (Robert) is present with substitution {Robert/P}. So we will add all the conjunctive facts below the first level and will replace p with Robert.

Here we can see American (Robert) is a fact, so it is proved here.

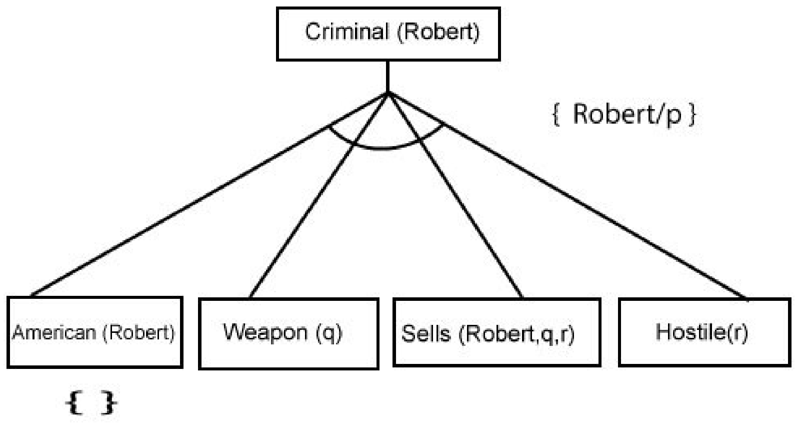


Figure 3.8

Step-3

t At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.

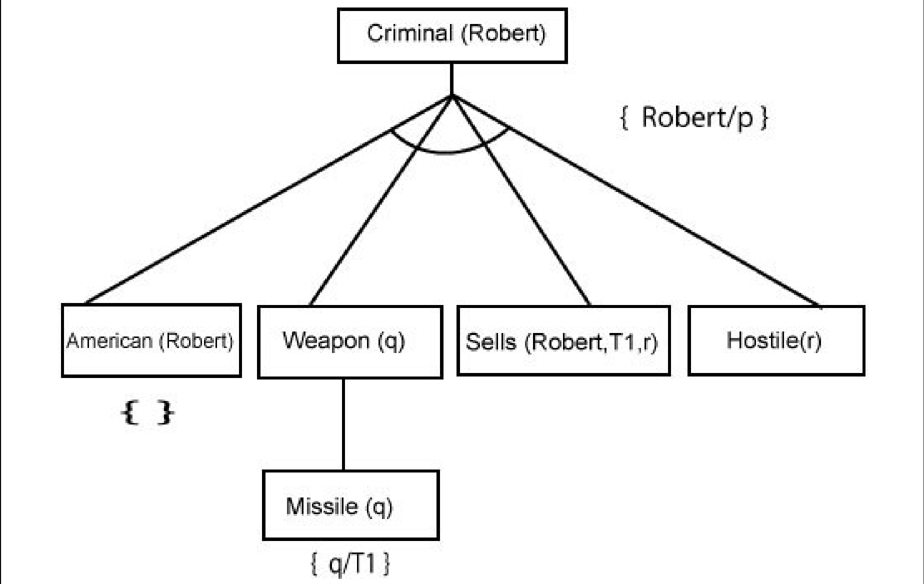


Figure 3.9

Step-4

At step-4, we can infer facts Missile(T1) and Owns(A, T1) form Sells(Robert, T1, r) which satisfies the Rule- 4, with the substitution of A in place of r. So these two statements are proved here.

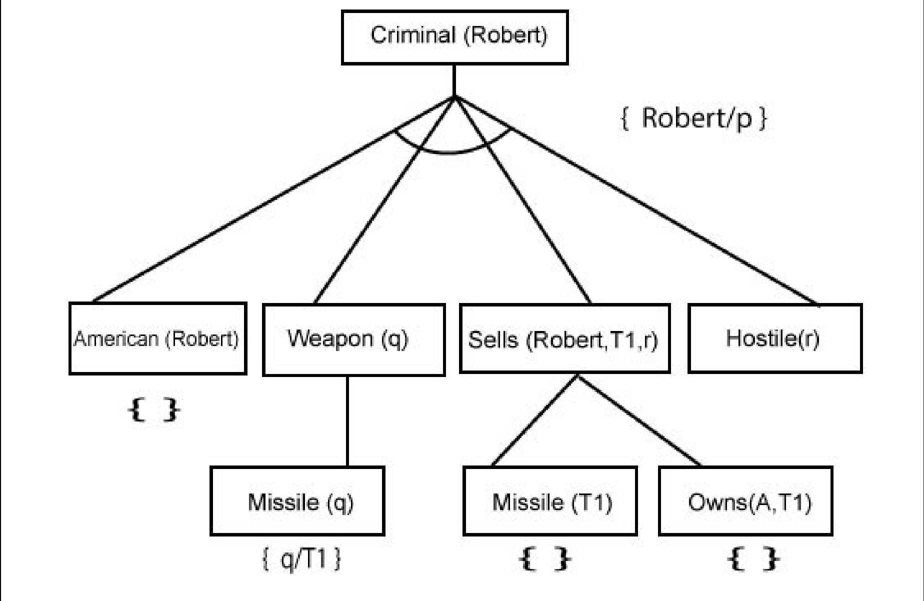


Figure 3.10

Step-5

At step-5, we can infer the fact Enemy(A, America) from Hostile(A) which satisfies Rule- 6. And hence all the statements are proved true using backward chaining.

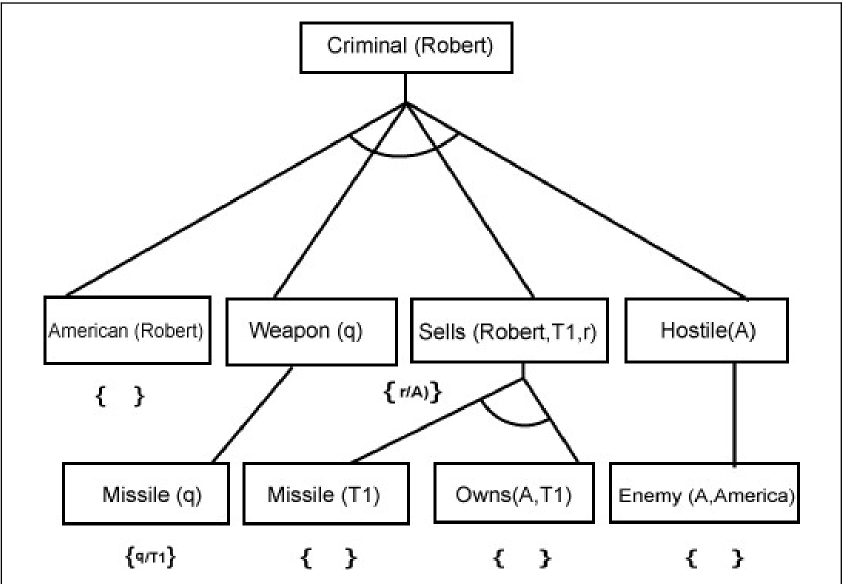


Figure 3.11

Suppose you have a production system with the FOUR rules: R1: IF A AND C then F R2: IF A AND E, THEN G R3: IF B, THEN E R4: R3: IF G, THEN D and you have four initial facts: A, B, C, D. PROVE A&B TRUE THEN D IS TRUE. Explain what is meant by "forward chaining", and show explicitly how it can be used in this case to determine new facts.

3.6 RESOLUTION IN FOL

Resolution

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. It was invented by a Mathematician John Alan Robinson in the year 1965.

Resolution is used, if there are various statements are given, and we 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 the conjunctive normal form or clausal form.

Clause: Disjunction of literals (an atomic sentence) is called a clause. It is also known as a unit clause.

Conjunctive Normal Form: A sentence represented as a conjunction of clauses is said to be conjunctive normal form or CNF.

Steps for Resolution

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph using Unification

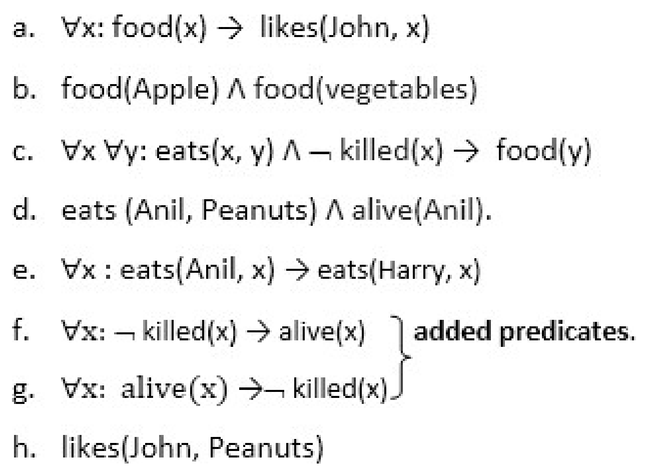
To better understand all the above steps, we will take an example in which we will apply resolution.

Example

a. John likes all kind of food.
b. Apple and vegetable are food
c. Anything anyone eats and not killed is food.
d. Anil eats peanuts and still alive
e. Harry eats everything that Anil eats. Prove by resolution that:
f. John likes peanuts.

Step-1: Conversion of Facts into FOL

In the first step we will convert all the given statements into its first order logic.



a. ∀x: food(x) → likes(John, x)
b. food(Apple) Λ food(vegetables)
c. ∀x ∀y: eats(x, y) Λ ¬ killed(x) → food(y)
d. eats (Anil, Peanuts) Λ alive(Anil).
e. ∀x : eats(Anil, x) → eats(Harry, x)
f. ∀x: ¬ killed(x) → alive(x)
g. ∀x: alive(x) →¬ killed(x)

added predicates.

h. likes(John, Peanuts)

- **Eliminate all implication (→) and rewrite**

1. ∀x ¬ food(x) V likes(John, x)
2. food(Apple) Λ food(vegetables)
3. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
4. eats (Anil, Peanuts) Λ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
6. ∀x¬ [¬ killed(x)] V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)

8. likes(John, Peanuts).

- **Move negation (¬)inwards and rewrite**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
  6. ∀x ¬killed(x)] V alive(x)
  7. ∀x ¬ alive(x) V ¬ killed(x)
  8. likes(John, Peanuts).
- **Rename variables or standardize variables**
  1. ∀x ¬ food(x) V likes(John, 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(Harry, w)
  6. ∀g ¬killed(g)] V alive(g)
  7. ∀k ¬ alive(k) V ¬ killed(k)
  8. likes(John, Peanuts).
- **Eliminate existential instantiation quantifier by elimination.** In this step, we will eliminate existential quantifier ∃, and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.
- **Drop Universal quantifiers.** In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.

  1. ¬ food(x) V likes(John, 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(Harry, w)
  8. killed(g) V alive(g)
  9. ¬ alive(k) V ¬ killed(k)
  10. likes(John, Peanuts).
- **Distribute conjunction ∧ over disjunction ¬.**This step will not make any change in this problem.

Step-3: Negate the statement to be proved

In this statement, we will apply negation to the conclusion statements, which will be written as ¬likes(John, Peanuts)

Step-4: Draw Resolution graph

- Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:

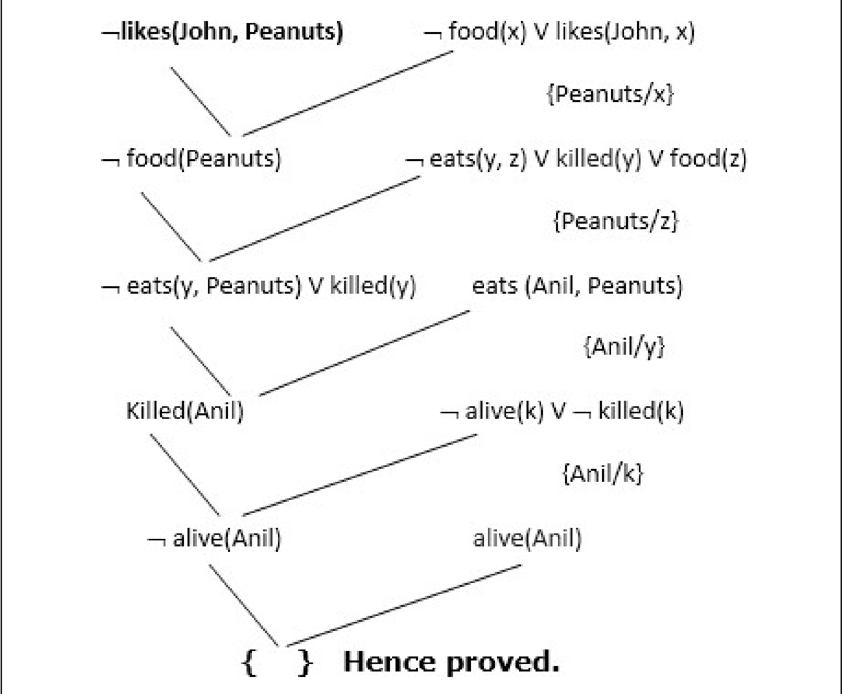


Figure 3.12

Explanation of Resolution graph

- In the first step of resolution graph, ¬likes(John, Peanuts), and likes(John, x) get resolved (canceled) by substitution of {Peanuts/x}, and we are left with ¬ food(Peanuts)
- In the second step of the resolution graph, ¬ food (Peanuts), and food(z) get resolved (canceled) by substitution of { Peanuts/z}, and we are left with ¬ eats(y, Peanuts) V killed(y).
- In the third step of the resolution graph, ¬ eats(y, Peanuts) and eats (Anil, Peanuts) get resolved by substitution {Anil/y}, and we are left with Killed(Anil).
- In the fourth step of the resolution graph, Killed(Anil) and ¬ killed(k) get resolve by substitution {Anil/k}, and we are left with ¬ alive(Anil).

o In the last step of the resolution graph ¬ alive(Anil) and alive(Anil) get resolved.

Difference between Predicate Logic and Propositional Logic

Table 3.1

S. No.	Predicate logic	Propositional logic
1.	Predicate logic is a generalization of propositional logic that allows us to express and infer arguments in infinite models.	A preposition is a declarative statement that's either TRUE or FALSE (but not both).
2.	Predicate logic (also called predicate calculus and first-order logic) is an extension of propositional logic to formulas involving terms and predicates. The full predicate logic is undecidable	Propositional logic is an axiomatization of Boolean logic. Propositional logic is decidable, for example by the method of truth table.
3.	Predicate logic have variables	Propositional logic has variables. Parameters are all constant.
4.	A predicate is a logical statement that depends on one or more variables (not necessarily Boolean variables)	Propositional logic deals solely with propositions and logical connectives.
5.	Predicate logic there are objects, properties, functions (relations) are involved.	Proposition logic is represented in terms of Boolean variables and logical connectives.
6.	In predicate logic, we symbolize subject and predicate separately. Logicians often use lowercase letters to symbolize subjects (or objects) and upper case letter to symbolize predicates.	In propositional logic, we use letters to symbolize entire propositions. Propositions are statements of the form "x is y" where x is a subject and y is a predicate.
7.	Predicate logic uses quantifies such as universal quantifier ("∀"), the existential quantifier ("∃").	Prepositional logic has not qualifiers.
8.	**Example** Everything is a green as "∀x Green(x)" or "Something is blue as "∃ x Blue(x)".	**Example** Everything is a green as "G" or "Something is blue as "B(x)".

<u>**EXAMPLE 1**</u>

Consider the crime example. The sentences in CNF are

¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x, y, z) ∨ ¬Hostile(z) ∨ Criminal (x)

¬Missile(x) ∨ ¬Owns(Nono, x) ∨ Sells(West, x, Nono)

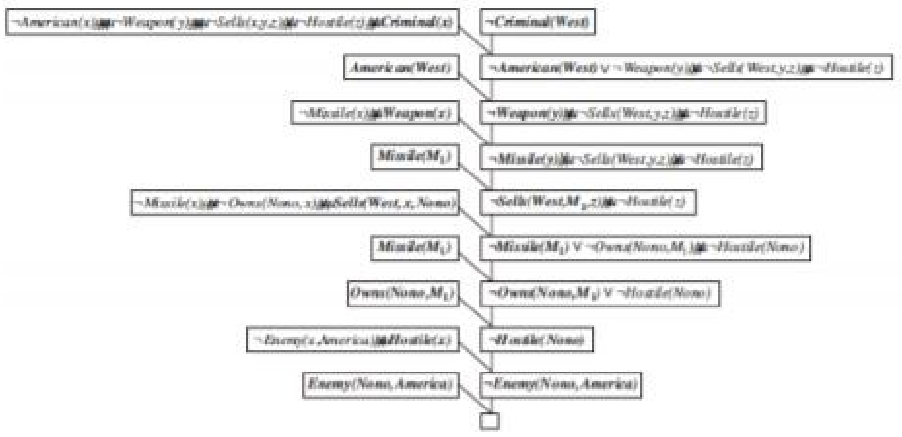
¬Enemy(x, America) ∨ Hostile(x)

¬Missile(x) ∨Weapon(x)

Owns(Nono, M1) Missile(M1)

American(West) Enemy(Nono, America)

We also include the negated goal ¬ Criminal (West). The resolution proof is shown in Figure.



¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x,y,z) ∨ ¬Hostile(z) ∨ Criminal(x)
¬Criminal(West)
American(West)
¬American(West) ∨ ¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ Weapon(x)
¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ ¬Owns(Nono,x) ∨ Sells(West,x,Nono)
¬Sells(West,M1,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(M1) ∨ ¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
Owns(Nono,M1)
¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
¬Enemy(x,America) ∨ Hostile(x)
¬Hostile(Nono)
Enemy(Nono,America)
¬Enemy(Nono,America)

Figure 4.7 A resolution proof that West is a criminal. At each step, the literals that unify are in bold.

Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond exactly to the consecutive values of the goals variable in the backward-chaining algorithm of Figure. This is because we always choose to resolve with a clause whose positive literal unified with the left most literal of the "current" clause on the spine; this is exactly what happens in backward chaining. Thus, backward chaining is just a special case of resolution with a particular control strategy to decide which resolution to perform next.

EXAMPLE 2

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is a follows:

Everyone who love all animals is loved by someone.
Anyone who kills an animals is loved by no one.
Jack loves all animals.

Either Jack or Curiosity killed the cat, who is named Tuna.

Did Curiosity kill the cat? First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. ∀ x [∀ y Animal(y) ⇒ Loves(x,y) ⇒ [∃y Loves(y,x)]
B. ∀ x [∃ x Animal(z) ^ Kills (x,z)] ⇒ [∀ y Loves(y,x)]
C. ∀ x Animals(x) ⇒Loves(Jack, x)
D. Kills (Jack, Tuna) V Kills (Curiosity, Tuna)
E. Cat (Tuna)
F. ∀ x Cat(x) ⇒ Animal (x)
¬G. ¬Kills(Curiosity, Tuna)

Now we apply the conversion procedure to convert each sentence to CNF:

A1. Animal(F(x)) v Loves (G(x),x)
A2. ¬Loves(x,F(x)) v Loves (G(x),x)
B. ¬Loves(y,x) v ¬ Animal(z) V ¬ Kills(x,z)
C. ¬Animal(x) v Loves(Jack, x)
D. Kills(Jack, Tuna) v Kills (Curiosity, Tuna)
E. Cat (Tuna)
F. ¬Cat(x) v Animal (x)
¬G. ¬Kills (Curiosity, Tuna)

The resolution proof that Curiosity kills the cat is given in Figure. In English, the proof could be paraphrased as follows:

Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore Curiosity killed the cat.

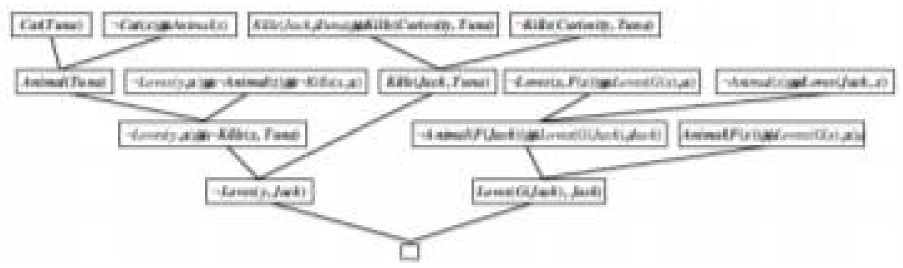


Figure 4.8 A resolution proof that Curiosity killed that Cat. Notice then use of factoring in the derivation of the clause Loves(G(Jack), Jack). Notice also in the upper right, the unification of Loves(x,F(x)) and Loves(Jack,x) can only succeed after the variables have been standardized apart

The proof answers the question "Did Curiosity kill the cat?" but often we want to pose more general questions, such as "Who killed the cat?" Resolution can do this, but it takes a little more work to obtain the answer. The goal is ∃w Kills (w, Tuna), which, when negated become ¬Kills (w, Tuna) in CNF, Repeating the proof in Figure with the new negated goal, we obtain a similar proof tree, but with the substitution {w/Curiosity} in one of the steps. So, in this case, finding out who killed the cat is just a matter of keeping track of the bindings for the query variables in the proof.

EXAMPLE 3

1. All people who are graduating are happy.
2. All happy people smile.
3. Someone is graduating.
4. Conclusion: Is someone smiling?

Solution

Convert the sentences into predicate Logic

1. ∀x graduating(x) -> happy(x)
2. ∀x happy(x) -> smile(x)
3. ∃x graduating(x)
4. ∃x smile(x)

Convert to clausal form

(i) Eliminate the → sign

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(x) vsmile(x)
3. ∃x graduating(x)
4. -∃x smile(x)

(ii) Reduce the scope of negation

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(x) vsmile(x)
3. ∃x graduating(x)
4. ∀x¬ smile(x)

(iii) Standardize variable apart

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(y) vsmile(y)
3. ∃x graduating(z)
4. ∀x¬ smile(w)

(iv) Move all quantifiers to the left

1. ∀x¬ graduating(x) vhappy(x)

2. ∀x¬ happy(y) vsmile(y)

3. ∃x graduating(z)

4. ∀w¬ smile(w)

(v) Eliminate ∃

1. ∀x¬ graduating(x) vhappy(x)
2. ∀x¬ happy(y) vsmile(y)
3. graduating(name1)
4. ∀w¬ smile(w)

(vi) Eliminate∀

1. ¬graduating(x) vhappy(x)
2. ¬happy(y) vsmile(y)
3. graduating(name1)
4. ¬smile(w)

(vii) Convert to conjunct of disjuncts form

(viii) Make each conjunct a separate clause.

(ix) Standardize variables apart again.

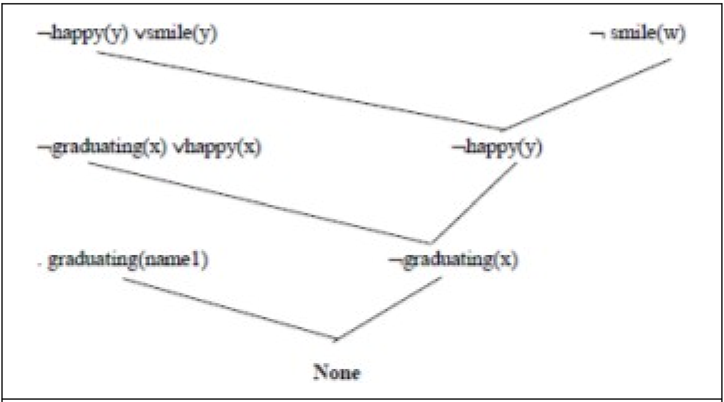


Figure 4.9 Standardize variables

Thus, we proved someone is smiling.

EXAMPLE 4

Explain the unification algorithm used for reasoning under predicate logic with an example. Consider the following facts

a. Team India

b. Team Australia

c. Final match between India and Australia

d. India scored 350 runs, Australia scored 350 runs, India lost 5 wickets, Australia lost 7 wickets.

f. If the scores are same the team which lost minimum wickets wins the match.

Represent the facts in predicate, convert to clause form and prove by resolution "India wins the match".

Solution

Convert into predicate Logic

(a) team(India)

(b) team(Australia)

(c) team(India) ^ team(Australia) → final_match(India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x team(x) ^ wins(x) → score(x,mat_runs)

(f) ∃xy score(x,equal(y)) ^ wicket(x,min) ^ final_match(x,y) → win(x)

Convert to clausal form

(i) Eliminate the → sign

(a) team(India)

(b) team(Australia)

(c) ¬(team(India) ^ team(Australia) v final_match(India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x¬ (team(x) ^ wins(x)) vscore(x,max_runs))

(f) ∃xy¬ (score(x,equal(y)) ^ wicket(x,min) ^final_match(x,y)) vwin(x)

(ii) Reduce the scope of negation

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match(India, Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x¬ team(x) v ¬wins(x) vscore(x,max_runs))

(f) ∃xy¬ (score(x,equal(y)) v ¬ wicket(x,min_wicket) v¬final_match(x,y)) vwin(x)

(iii) Standardize variables apart

(iv) Move all quantifiers to the left

(v) Eliminate ∃

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match (India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ¬team(x) v ¬wins(x) vscore(x, max_runs))

(f) ¬score(x,equal(y)) v¬wicket(x,min_wicket) v-final_match(x,y)) vwin(x)

(vi) Eliminate∀

(vii) Convert to conjunct of disjuncts form.

(viii) Make each conjunct a separate clause.

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match (India,Australia)

(d) score(India,350)
Score(Australia,350)
Wicket(India,5)
Wicket(Austrialia,7)

(e) ¬team(x) v ¬wins(x) vscore(x,max_runs))

(f) ¬score(x,equal(y)) v¬wicket(x,min_wicket) v¬final_match(x,y)) vwin(x)

(ix) Standardize variables apart again

To prove:win(India) **Disprove:**¬win(India)

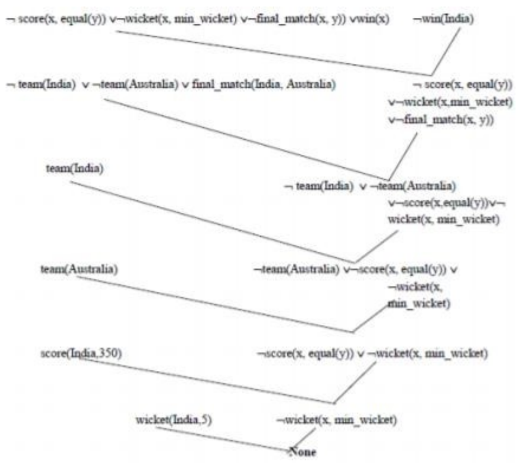

¬ score(x, equal(y)) v¬wicket(x, min_wicket) v¬final_match(x, y)) vwin(x)
¬win(India)
¬ team(India) v ¬team(Australia) v final_match(India, Australia)
¬ score(x, equal(y)) v¬wicket(x,min_wicket) v¬final_match(x, y))
team(India)
¬ team(India) v ¬team(Australia) v¬score(x,equal(y))v¬ wicket(x, min_wicket)
team(Australia)
¬team(Australia) v¬score(x, equal(y)) v ¬wicket(x, min_wicket)
score(India,350)
¬score(x, equal(y)) v ¬wicket(x, min_wicket)
wicket(India,5)
¬wicket(x, min_wicket)
None

Figure 4.10 Standardize variables

Thus, proved India wins match.

EXAMPLE 5

Problem 3

Consider the following facts and represent them in predicate form:

F1. There are 500 employees in ABC company.

F2. Employees earning more than Rs. 5000 per tax.

F3. John is a manger in ABC company.

F4. Manger earns Rs. 10,000.

Convert the facts in predicate form to clauses and then prove by resolution: “John pays tax”.

Solution

Convert into predicate Logic

1. company(ABC) ^employee(500,ABC)
2. ∃x company(ABC) ^employee(x,ABC) ^ earns(x,5000)→pays(x,tax)
3. manager(John,ABC)
4. ∃x manager(x, ABC)→earns(x,10000)

Convert to clausal form

(i) Eliminate the → sign

1. company(ABC) ^employee(500,ABC)
2. ∃x¬(company(ABC) ^employee(x,ABC) ^ earns(x,5000)) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

(ii) Reduce the scope of negation

1. company(ABC) ^employee(500,ABC)
2. ∃x¬ company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

(iii) Standardize variables apart

1. company(ABC) ^employee(500,ABC)
2. ∃x¬ company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

(iv) Move all quantifiers to the left

(v) Eliminate ∃

1. company(ABC) ^employee(500,ABC)
2. ¬company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ¬manager(x, ABC) v earns(x,10000)

(vi) Eliminate∀

(vii) Convert to conjunct of disjuncts form

(viii) Make each conjunct a separate clause.

1. (a) company(ABC)
   (b) employee(500,ABC)
2. ¬company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ¬manager(x, ABC) v earns(x,10000)

(ix) Standardize variables apart again.

Prove:pays(John,tax)

Disprove:¬pays(John,tax)

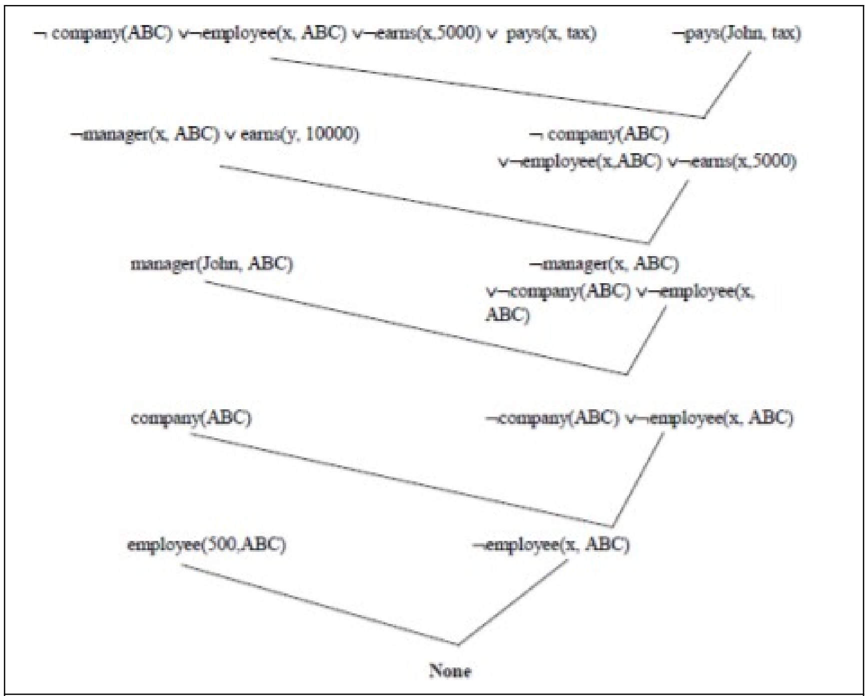

¬ company(ABC) v¬employee(x, ABC) v¬earns(x,5000) v pays(x, tax)
¬pays(John, tax)
¬manager(x, ABC) v earns(y, 10000)
¬ company(ABC) v¬employee(x,ABC) v¬earns(x,5000)
manager(John, ABC)
¬manager(x, ABC) v¬company(ABC) v¬employee(x, ABC)
company(ABC)
¬company(ABC) v¬employee(x, ABC)
employee(500,ABC)
¬employee(x, ABC)
None

Figure 4.11 Standardize variables

Thus, proved john pays tax.

EXAMPLE 6 & EXAMPLE 7

Problem 4

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.

Convert into predicate Logic

1. ∀xyperfect_sq(x) ^ prime(h) ^divideds(x,y) → divides(x,square(y))

2. ∀x∃y perfect)sq(x) ^prime(y) ^divides(x,y)

3. perfect_sq(36)

Problem 5

1. Marcus was a a man

   man(Marcus)

2. Marcus was a Pompeian

   Pompeian(Marcus)

3. All Pompeians were Romans

   ∀x (Pompeians(x) → Roman(x))

4. Caesar was ruler

   Ruler(Caesar)

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

   ∃x (Roman(x) → loyalto(x,Caesar) v hate(x,Caesar))

6. Everyone is loyal to someone

   ∀x ∃y (person(x) → person(y) ^ loyalto(x,y))

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

   ∀x ∃y (person(x) ^ ruler(y)^ tryassassinate(x,y) → -loyalto(x,y))

8. Marcus tried to assassinate Caesar

   tryassassinate(Marcus, Caesar)

9. All men are persons

   ∀x (max(x) → person(x))

The Unification algorithm is listed below as a procedure UNIFY (L1, L2). It returns a list representing the composition of the substitutions that were performed during the match. An empty list NIL indicates that a match was found without any substitutions. If the list contains a single value F, it indicates that the unification procedure failed.

UNIFY (L1, L2)

1. if L1 or L2 is an atom part of same thing do

(a) if L1 or L2 are identical then return NIL

(b) else if L1 is a variable then do

(i) if L1 occurs in L2 then return F else return (L2/L1)

© else if L2 is a variable then do

(i) if L2 occurs in L1 then return F else return (L1/L2)

else return F.

2. If length (L!) is not equal to length (L2) then return F.

3. Set SUBST to NIL

(at the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2).

4. For I = 1 to number of elements in L1 do

i) call UNIFY with the ith element of L1 and I'th element of L2, putting the result in S

ii) if S = F then return F

iii) if S is not equal to NIL then do

(A) apply S to the remainder of both L1 and L2

(B) SUBST := APPEND (S, SUBST) return SUBST.

Sample questions

1)Consider a knowledge base containing just two sentences: P(a) and P(b). Does this knowledge base entail ∀ x P(x)? Explain your answer in terms of models.

The knowledge base does not entail ∀*x* *P*(*x*). To show this, we must give a model where *P*(*a*) and *P*(*b*) but ∀*x* *P*(*x*) is false. Consider any model with three domain elements, where *a* and *b* refer to the first two elements and the relation referred to by *P* holds only for those two elements.

2)Is the sentence ∃ x, y x = y valid? Explain

The sentence ∃ *x, y x*=*y* is valid. A sentence is valid if it is true in every model. An existentially quantified sentence is true in a model if it holds under any extended interpretation in which its variables are assigned to domain elements. According to the standard semantics of FOL as given in the chapter, every model contains at least one domain element, hence, for any model, there is an extended interpretation in which *x* and *y* are assigned to the first domain element. In such an interpretation, *x*=*y* is true.

3)Differentiate between propositional and first order predicate logic?

Following are the comparative differences versus first order logic and propositional logic.

1) Propositional logic is less expressive and do not reflect individual object`s properties explicitly. First order logic is more expressive and can represent individual object along with all its properties.

2) Propositional logic cannot represent relationship among objects whereas first order logic can represent relationship.

3) Propositional logic does not consider generalization of objects where as first order logic handles generalization.

4) Propositional logic includes sentence letters (A, B, and C) and logical connectives, but not quantifier. First order logic has the same connectives as

propositional logic, but it also has variables for individual objects, quantifier, symbols for functions and symbols for relations.

4)Represent the following sentence in predicate form:

"All the children like sweets"

$\forall x$ child(x) $\cap$ sweet(y) $\cap$ likes (x,y).

5)Illustrate the use of first order logic to represent knowledge.

The best way to find usage of First order logic is through examples. The examples can be taken from some simple domains. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge. Assertions and queries in first-order logic Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called assertions. For example, we can assert that John is a king and that kings are persons: Where KB is knowledge base. TELL(KB, $\forall x$ King(x) => Person(x)). We can ask questions of the knowledge base using AS K. For example, returns true. Questions asked using ASK are called queries or goals ASK(KB, Person(John)) Will return true. (ASK KBto find whether Jon is a king) ASK (KB, $\exists x$ person(x)) The kinship domain The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William7' and rules such as "One's grandmother is the mother of one's parent." Clearly, the objects in our domain are people. We will have two unary predicates, Male and Female. Kinship relations-parenthood, brotherhood, marriage, and so on-will be represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son, Spouse, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle. We will use functions for Mother and Father.