Module 4

First Order Logic and Inferences in FOL

Contents

1. First Order Logic:
    a. **Representation Revisited,**
    b. **Syntax and Semantics of First Order Logic,**
    c. **Using First Order Logic**.
2. Inference in First Order Logic:
    a. **Propositional Versus First Order Inference,**
    b. **Unification,**
    c. **Forward Chaining,**

Types of logic

Logics are characterized by what they commit to as "primitives"

Ontological commitment: what exists—facts? objects? time? beliefs?

Epistemological commitment: what states of knowledge?

Language	Ontological Commitment	Epistemological Commitment
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 0...1
Fuzzy logic	degree of truth	degree of belief 0...1

Propositional Logic

- Propositional logic, also known as **sentential logic or propositional calculus**, is a branch of formal logic that deals with the logical relationships between **propositions** (statements or sentences) without considering the **internal structure of the propositions.**
- In propositional logic, **propositions are considered as atomic units**, and logical operations are applied to these propositions to **form more complex statements**.

Some key elements and concepts in propositional logic:

1. **Propositions:** These are statements that can be either true or false. Propositions are represented by variables, typically denoted by letters (p, q, r, etc.).
2. **Logical Connectives:** These are symbols that combine propositions to form more complex statements. The main logical connectives in propositional logic include:
    1. **Conjunction (∧):** Represents "and." The compound proposition "p ∧ q" is true only when both p and q are true.
    2. **Disjunction (∨):** Represents "or." The compound proposition "p ∨ q" is true when at least one of p or q is true.
    3. **Negation (¬):** Represents "not." The compound proposition "¬p" is true when p is false.
    4. **Implication (→):** Represents "if...then." The compound proposition "p → q" is false only when p is true and q is false.
    5. **Biconditional (↔):** Represents "if and only if." The compound proposition "p ↔ q" is true when p and q have the same truth value.
3. **Truth Tables:** Truth tables are used to systematically list all possible truth values for a compound proposition based on the truth values of its constituent propositions. Truth tables help determine the truth conditions for complex statements.
4. **Logical Equivalence:** Two propositions are logically equivalent if they have the same truth values for all possible combinations of truth values of their component propositions.

$$
\begin{aligned}
\mathit{Sentence} &\rightarrow \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} &\rightarrow \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} &\rightarrow (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
&\mid \neg\ \mathit{Sentence} \\
&\mid \mathit{Sentence} \wedge \mathit{Sentence} \\
&\mid \mathit{Sentence} \vee \mathit{Sentence} \\
&\mid \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
&\mid \mathit{Sentence} \Leftrightarrow \mathit{Sentence}
\end{aligned}
$$

$$
\text{OPERATOR PRECEDENCE} \quad : \quad \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Propositional logic

- **Example.** We have the following sentences:
  - P: "Humidity is high"
  - Q: "Temperature is high"
  - C: "One feels comfortable".
- Then the sentence "If the humidity is high and the temperature is high, then one does not feel comfortable" may be represented by $((P \wedge Q) \Rightarrow (\sim C))$.

Propositions Examples-

The examples of propositions are-

- 7 + 4 = 10
- Apples are black.
- Narendra Modi is president of India.
- Two and two makes 5.
- 2016 will be the lead year.
- Delhi is in India.

The examples of atomic propositions are-

- p : Sun rises in the east.
- q : Sun sets in the west.
- r : Apples are red.
- s : Grapes are green.

Compound Propositions-

- P : Sun rises in the east and Sun sets in the west.
- Q : Apples are red and Grapes are green.

Statements That Are Not Propositions-

Following kinds of statements are not propositions-

- Command
- Question
- Exclamation
- Inconsistent
- Predicate or Proposition Function

Following statements are not propositions-

- Close the door. (Command)
- Do you speak French? (Question)
- What a beautiful picture! (Exclamation)
- I always tell lie. (Inconsistent)
- P(x) : x + 3 = 5 (Predicate)

Properties of PL

1. Propositional logic is a **declarative language** because its semantics is based on a truth relation between sentences and possible worlds.
2. It also has sufficient **expressive power to deal with partial information**, using **disjunction and negation**.
3. Propositional logic is a **compositional language**, where in a sentence is a function of the meaning of its parts.
   - For example, the meaning of "**S1,4 ∧ S1,2**" is related to the meanings of "**S1,4**" and "**S1,2**."

Drawbacks of Propositional Logic

- We can only represent information as either **true or false** in propositional logic.
- Expressive power of Propositional logic is **very limited** and lacks to describe an environment with many objects
- If you want to represent complicated sentences or natural language statements, **PL is not sufficient.**
- **Examples: PL is not enough to represent the sentences below**, so we require powerful logic (such as FOL).
  1. I love mankind. It's the people I can't stand!
  2. I like to eat mangos.

What is First Order Logic (FOL)?

1. FOL is also called ***predicate logic***. A much more expressive language than the propositional logic. It is a powerful language used to develop information about an **object and express the relationship** between objects.
2. FOL not only assumes that does the world contains facts (like PL does), but it also assumes the following:
    1. **Objects**: A, B, people, numbers, colors, wars, theories, squares, pit, etc.
    2. **Relations**: It is unary relation such as red, round, sister of, brother of, etc.
    3. **Function**: father of, best friend, third inning of, end of, etc.

First Order Logic Sentences

For each of the following English sentences, write a corresponding sentence in FOL.

1. The only good extraterrestrial is a drunk extraterrestrial.
   $\forall x.ET(x) \wedge Good(x) \rightarrow Drunk(x)$

2. The Barber of Seville shaves all men who do not shave themselves.
   $\forall x.\neg Shaves(x,x) \rightarrow Shaves(BarberOfSeville,x)$

3. There are at least two mountains in England.
   $\exists x,y.Mountain(x) \wedge Mountain(y) \wedge InEngland(x) \wedge InEngland(y) \wedge x \neq y$

4. There is exactly one coin in the box.
   $\exists x.Coin(x) \wedge InBox(x) \wedge \forall y.(Coin(y) \wedge InBox(y) \rightarrow x = y)$

5. There are exactly two coins in the box.

$$\exists x, y. Coin(x) \wedge InBox(x) \wedge Coin(y) \wedge InBox(y) \wedge x \neq y \wedge \forall z. (Coin(z) \wedge InBox(z) \rightarrow (x = z \vee y = z))$$

6. The largest coin in the box is a quarter.

$$\exists x. Coin(x) \wedge InBox(x) \wedge Quarter(x) \wedge \forall y. (Coin(y) \wedge InBox(y) \wedge \neg Quarter(y) \rightarrow Smaller(y, x))$$

7. No mountain is higher than itself.

$$\forall x. Mountain(x) \rightarrow \neg Higher(x, x)$$

8. All students get good grades if they study.

$$\forall x. Student(x) \wedge Study(x) \rightarrow GetGoodGrade(x)$$

Objects Relations and Functions

- **Objects:** people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries ...
- **Relations**: these can be unary relations or properties such as red, round, bogus, prime, multistoried ..., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, ...
- **Functions**: father of, best friend, third inning of, one more than, beginning of ..

Examples

- “**One plus two equals three**.”
  - **Objects**: one, two, three, one plus two;
  - **Relation**: equals;
  - **Function**: plus. (“One plus two” is a name for the object that is obtained by applying the function “plus” to the objects “one” and “two.” “Three” is another name for this object.)
- “**Squares neighboring the wumpus are smelly**.”
  - **Objects**: wumpus, squares;
  - **Property**: smelly;
  - **Relation**: neighboring.
- “**Evil King John ruled England in 1200**.”
  - **Objects:** John, England, 1200;
  - **Relation**: ruled;
  - **Properties**: evil, king.

Basic Elements of FOL

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >, <,Sister, Father.....
Function	sqrt, LeftLegOf, Sqrt, LessThan, Sin(θ).....
Connectives	∧, ∨, ¬, ⇒, ⇔
Equality	==
Quantifier	∀, ∃

Syntax and Semantics of FOL

1. **Models** for first-order logic
2. **Symbols** and interpretations
3. **Terms**
4. **Atomic** sentences
5. **Complex** sentences
6. **Quantifiers**: Universal quantification **(∀)** /Existential quantification **(∃)**
7. **Equality**
8. **An alternative semantics?** : **Data base Semantics**

Models for FOL (Key Characteristics)

A model in first-order logic is an interpretation that specifies: what each predicate means and the entities that can instantiate the variables.

- They have **objects** in them!
- The **domain** of a **model is 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
- The objects in the model **may be related** in various ways.
- Models in first-order logic require **total functions, that is, there must be a value for every input tuple**

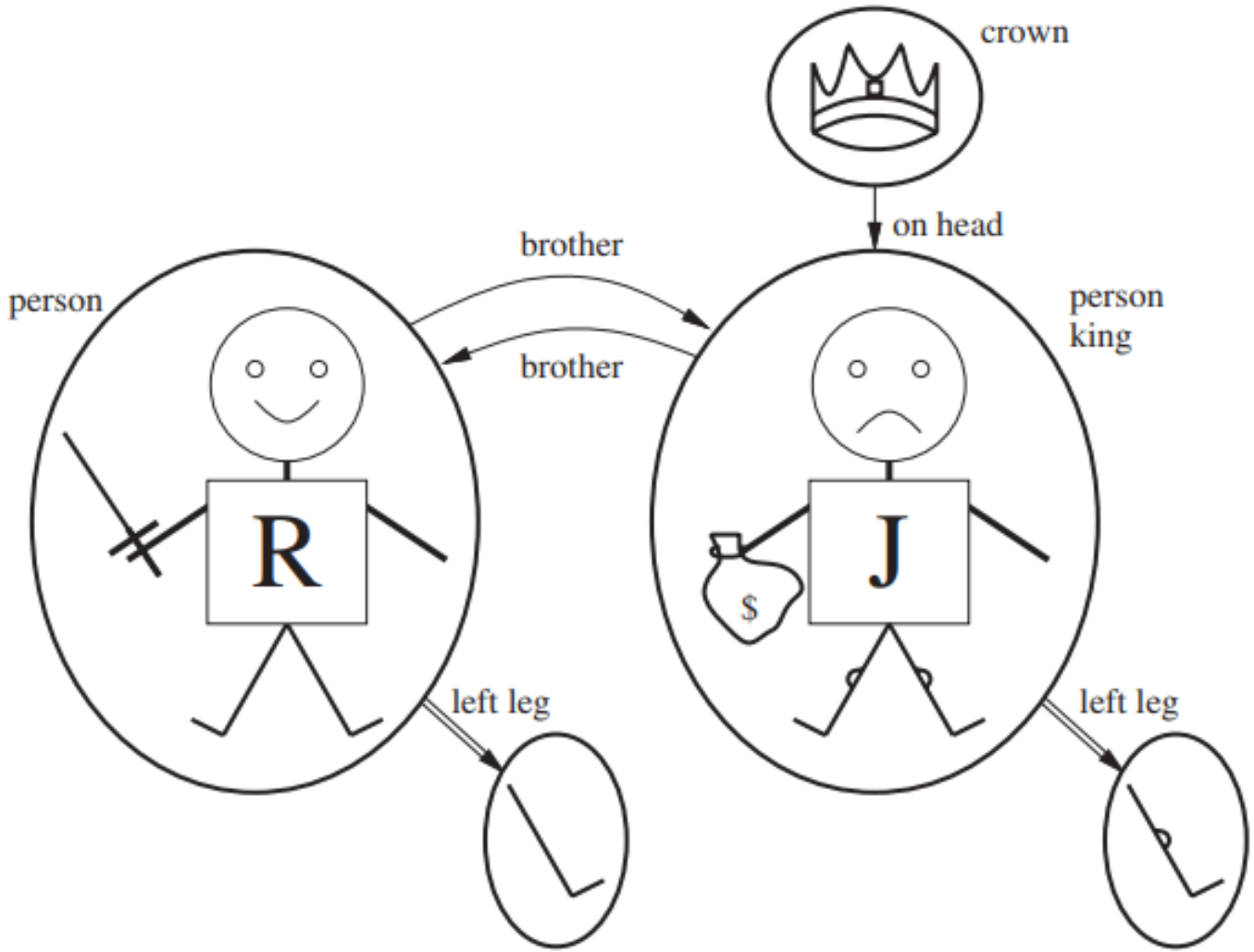

Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

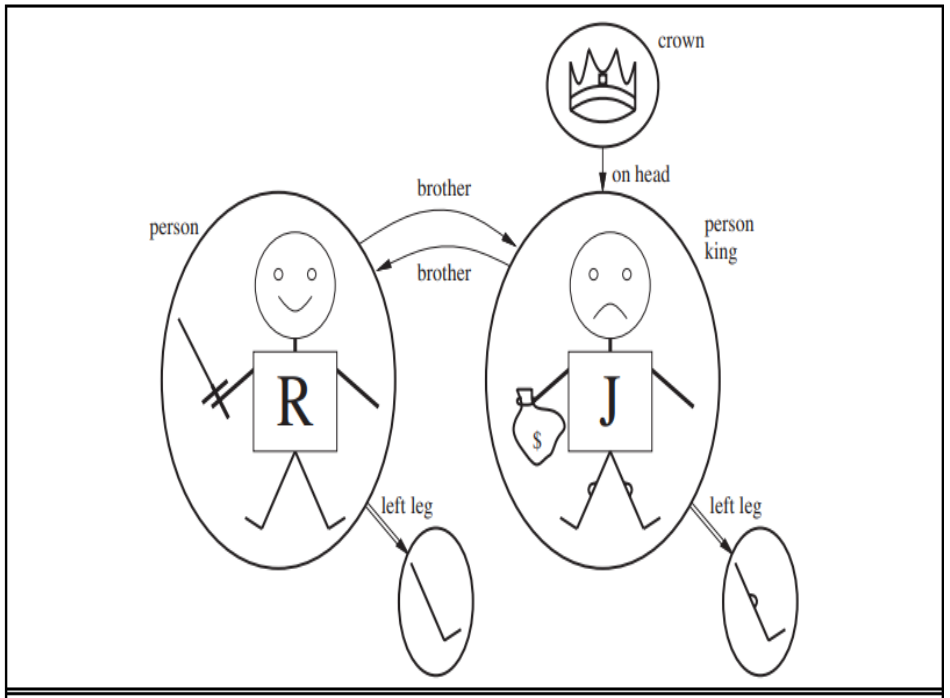

Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

Five objects:

1. Richard the Lionheart, King of England from 1189 to 1199;
2. His younger brother, the evil King John, who ruled from 1199 to 1215;
3. The left legs of Richard and John; and
4. Crown

Tuple : The brotherhood relation in this model is the set { <Richard the Lionheart, King John>, <King John, Richard the Lionheart> } .

Two binary relations : “brother” and “on head” relations are binary relations

Three unary relations/ properties : Person, King and Crown

One unary Function: Left Leg

Syntax of FOL

- **Symbols:** The basic syntactic elements **of first-order logic** are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds:
  1. **Constant symbols**, which stand for objects;
  2. **Predicate symbols**, which stand for relations; and
  3. **Function symbols**, which stand for functions.
- **Convention** : Symbols will begin with uppercase letters.
- Example
  - Constant symbols Richard and John;
  - Predicate symbols Brother , OnHead, Person, King, and Crown; and
  - the function symbol LeftLeg.
- **Arity** : Each predicate and function symbol comes with an arity that fixes the number of arguments

Syntax of FOL

- **Interpretation**: specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.
- Examples :
  - **Richard** refers to Richard the Lionheart
  - **John** refers to the evil King John.
  - **Brother** refers to the brotherhood relation
  - **OnHead** refers to the “on head” relation that holds between the crown and King John;
  - **Person, King, and Crown** refer to the sets of objects that are persons, kings, and crowns
  - **LeftLeg** refers to the “left leg” function

The syntax of first-order logic with equality, specified in Backus–Naur form

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} &\rightarrow (\ \textit{Sentence}\ ) \mid [\ \textit{Sentence}\ ] \\
&\mid \neg\ \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
&\mid \textit{Quantifier}\ \textit{Variable}, \ldots\ \textit{Sentence} \\
\textit{Term} &\rightarrow \textit{Function}(\textit{Term}, \ldots) \\
&\mid \textit{Constant} \\
&\mid \textit{Variable} \\
\textit{Quantifier} &\rightarrow \forall \mid \exists \\
\textit{Constant} &\rightarrow A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\textit{Predicate} &\rightarrow \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\textsc{Operator Precedence} &: \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

Syntax and Semantics of FOL

1. Models for first-order logic
2. Symbols and interpretations
3. Terms
4. Atomic sentences
5. Complex sentences
6. Quantifiers: Universal quantification (∀) /Existential quantification (∃)
7. Equality
8. An alternative semantics? : Data base Semantics

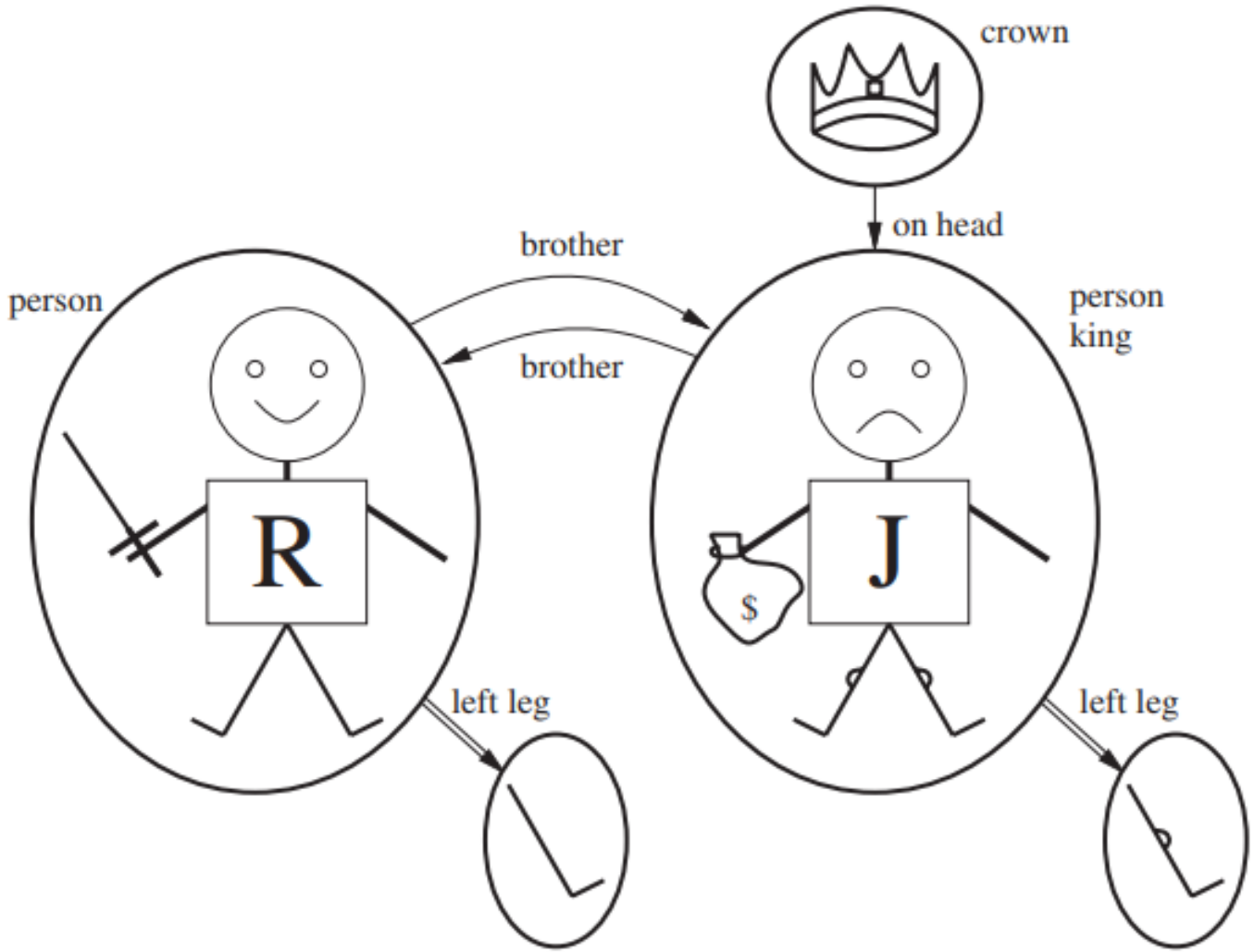

Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

3.Terms

- A term is a logical expression that refers to an object. Constant symbols are terms.
- It is not always convenient to have a distinct symbol to name every object.
  - **Example** : "**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 **LeftLeg(John).**
- In the general case, a complex term is formed by a **function symbol** followed by a **parenthesized** list of terms as arguments to the function symbol.(It is not a subroutine or function call)

3.Terms

- **Formal Semantics** : **Consider a term f(t1,... ,tn).**
- The function symbol **f r**efers to some function in the model (call it **F**); the argument terms refer to objects in the domain (call them d1,... ,dn);
- **Example** : LeftLeg(John):
- The **LeftLeg** function symbol refers to the function and **John** refers to King John, then **LeftLeg(John)** refers to **King John’s left leg.**

4.Atomic sentences

- An atomic sentence (**or atom for short**) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such a
  - ***Brother (Richard, John) :*** This states, that Richard the Lionheart is the brother of King John.

- Atomic sentences can have **complex terms as arguments**:
  - **Married(Father (Richard), Mother (John))** : states that Richard the Lionheart's father is married to King John's mother

5.Complex Sentences

- We can use **logical connectives to construct more complex sentences**, with the same syntax and semantics as in propositional calculus
- **Example** : Here are four sentences that are true in the model
  1. **¬Brother (LeftLeg(Richard), John)**
  2. **Brother (Richard, John) ∧ Brother (John, Richard)**
  3. **King(Richard) ∨ King(John)**
  4. **¬King(Richard) ⇒ King(John)**

6. Quantifiers

- **Quantifiers are essential** for expressing statements about collections of objects or individuals in a precise and concise manner within **first-order logic**.
- There are two main quantifiers:
  1. **the existential quantifier (∃)** and
  2. **the universal quantifier (∀).**

6. Quantifiers: Universal quantification (∀)

Universal Quantifier (∀): Denoted by the symbol "∀".

- It asserts that a **predicate or condition is true for all** instances of a variable in a given domain.
- For example, the statement **"∀x P(x)"** asserts that the predicate **P(x)** is true for all x in the domain.
- By convention, variables are lowercase letters.
- 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.**

6. Universal Quantifiers: Examples

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

6. Universal Quantifiers: Examples

- 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:

1. **Richard the Lionheart is a king ⇒ Richard the Lionheart is a person.**
2. **King John is a king ⇒ King John is a person.**
3. **Richard's left leg is a king ⇒ Richard's left leg is a person.**
4. **John's left leg is a king ⇒ John's left leg is a person.**
5. **The crown is a king ⇒ the crown is a person**

6. Universal Quantifiers: Examples

- A **common mistake, made frequently** even by diligent readers who have read this paragraph several times, is to use conjunction instead of implication.
- The sentence **∀ x King(x) ∧ Person(x**) would be equivalent to asserting

1. Richard the Lionheart is a king ∧ Richard the Lionheart is a person,
2. King John is a king ∧ King John is a person,
3. Richard's left leg is a king ∧ Richard's left leg is a person,

6. Quantifiers: Existential quantification (∃)

- Denoted by the symbol **"∃".**
- It asserts that there exists **at least one instance** of a variable that satisfies a given predicate or condition.
- **For example,** the statement **"∃x P(x)"** asserts that there exists at least one x such that the predicate P(x) is true.

6. Quantifiers: Existential quantification (∃)

- 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...”

6. Quantifiers: Existential quantification (∃)

- Intuitively, the sentence **∃ x P says that P is true** for at least one **object x.**
- **That is, at least one of the following is true:**

1. Richard the Lionheart is a crown ∧ Richard the Lionheart is on John's head;
2. King John is a crown ∧ King John is on John's head;
3. Richard's left leg is a crown ∧ Richard's left leg is on John's head;
4. John's left leg is a crown ∧ John's left leg is on John's head;
5. The crown is a crown ∧ the crown is on John's head

6. Quantifiers: Existential quantification (∃)

- 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:

1. Richard the **Lionheart is a crown ⇒ Richard the Lionheart is on John's head**;
2. **King John is a crown ⇒ King John is on John's head**;
3. **Richard's left leg is a crown ⇒ Richard's left leg is on John's head**;

6. 1 : Nested Quantifiers

- Nested quantifiers in **First-Order Logic (FOL)** refer to situations where quantifiers are used within the scope of other quantifiers in a logical expression.
- This nesting allows for the expression of more complex relationships and properties involving multiple variables.
- The universal quantifier (∀) and the existential quantifier (∃), both can be used in nested configurations.

6. 1 : Nested Quantifiers : Examples and Interpretations

1. **$\forall x \exists y P(x,y)$:**
   - This statement means "**for every x, there exists a y such that the property P(x,y) holds**."
   - It asserts a universal condition on x and, for each x, an existential condition on y.
2. **$\exists x \forall y P(x,y)$:**
   - This statement means "**there exists an x such that for every y, the property P(x,y) holds**."
   - It asserts the existence of a particular x for which a universal statement about y is true.

6. 1 : Nested Quantifiers : Examples and Interpretations

- **Importance of Order :** The order of **nested quantifiers is crucial** because it can change the meaning of a statement.
- For instance, the two examples given above have significantly different meanings due to the order of quantification.
- In general, **changing the order of quantifiers** in a statement with nested quantifiers will result in a statement that expresses a different property or relationship.

6. 1 : Nested Quantifiers : Examples and Interpretations

- Example: "**for every natural number, there exists a prime number greater than it**"
- $(\forall x \in \mathbb{N}, \exists y\ (y > x \wedge \text{Prime}(y)))$

6. 1 : Nested Quantifiers : Examples and Interpretations

- “Everybody loves somebody” : **∀ x ∃ y Loves(x,y)**
- “There is someone who is loved by everyone,”: **∃ 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**

6. 1 : Nested Quantifiers : 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)**

6. 1 : Nested Quantifiers : Connections between ∀ and ∃

- "**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) .

6. 2 : De Morgans Rules for quantified and unquantified sentences

- $\forall x \neg P \equiv \neg \exists x P$
- $\neg \forall x P \equiv \exists x \neg P$
- $\forall x P \equiv \neg \exists x \neg P$
- $\exists x P \equiv \neg \forall x \neg P$

$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$

$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$

$P \wedge Q \equiv \neg(\neg P \vee \neg Q)$

$P \vee Q \equiv \neg(\neg P \wedge \neg Q)$.

7.Equality

- In First-Order Logic (FOL), equality is a fundamental concept that allows the expression of the notion that two terms denote the same object.
- **Syntax**: In the syntax of FOL, an equality statement typically looks like a = b where ***a* and *b* are terms in the logic**. Terms can be variables, constants, or any expression that refers to objects in the domain of discourse.
- **Semantics**: The semantics of equality states that a = b is true if and only if *a* and *b* refer to the **same object** in the domain of discourse.

7.Equality: Examples

- **Father (John)= Henry** says that the object referred to by Father (John) and the object referred to by Henry are the same.
- To say that Richard has at least two brothers, we would write
- **∃ x,y Brother (x, Richard) ∧ Brother (y, Richard) ∧ ¬(x = y) .**

8.Any Alternative Semantics : Database Semantics

- One proposal that is very popular in database systems works as follows.
- **First, we** insist that every **constant symbol** refer to a distinct object—the so-called **unique-names assumption**.
- **Second,** we assume that atomic sentences not known to be true are **in fact false—the closed-world assumption**.
- Finally, we invoke **domain closure**, meaning that each model contains no more domain elements than those named by the **constant symbols**.

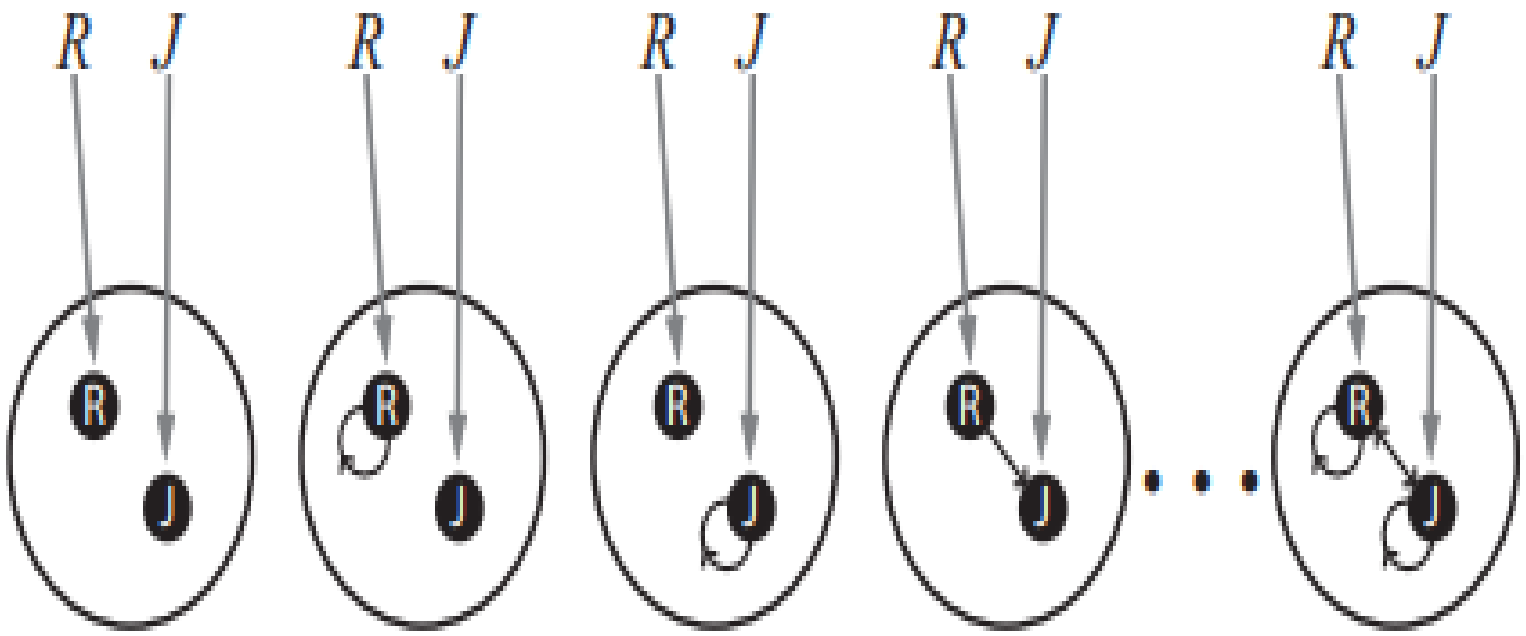

Figure 8.5 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

4.1.c Using First Order Logic

- **In this section, we discuss 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.**

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**
- We can ask questions of the knowledge base using **ASK**. Questions asked with ASK are called **queries or goals.**

Examples

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

ASK(KB, King(John))
ASK(KB, Person(John))
ASK(KB, ∃ x Person(x)) .

ASKVARS : Substitution or binding List

- **ASKVARS(KB, Person(x))** 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. It** will bind the variables to specific values.

- **Note: i**f KB has been told **King(John) ∨ King(Richard),** then there is **no binding to x for the query ∃ x King(x)**, even though the query is true.

Example: The domain of family relationships, or kinship domain

- **This domain includes facts such as**
  - **“Elizabeth is the mother of Charles”** and
  - **“Charles is the father of William”** and rules such as
  - **“One’s grandmother is the mother of one’s parent**.”
- Clearly, the objects in our domain are people. We have two unary predicates, **Male and Female**.
- **Kinship relations**—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: ***Parent, Sibling, Brother , Sister , Child, Daughter , Son, Spouse, Wife, Husband, Grandparent, Grandchild , Cousin, Aunt, and Uncle.***

- **One's mother is one's female parent**: $\forall m,c\ Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m,c)$.
- **One's husband is one's male spouse**: $\forall w,h\ Husband(h,w) \Leftrightarrow Male(h) \wedge Spouse(h,w)$.
- **Male and female are disjoint categories**: $\forall x\ Male(x) \Leftrightarrow \neg Female(x)$.
- **Parent and child are inverse relations**: $\forall p,c\ Parent(p,c) \Leftrightarrow Child(c,p)$
- **A grandparent is a parent of one's parent**:

  $\forall g,c\ Grandparent(g,c) \Leftrightarrow \exists p\ Parent(g,p) \wedge Parent(p,c)$.
- **A sibling is another child of one's parents:**

  $\forall x,y\ Sibling(x,y) \Leftrightarrow x \neq y \wedge \exists p\ Parent(p,x) \wedge Parent(p,y)$.

Axioms : Each of these sentences can be viewed as an axiom of the kinship domain. They provide the basic factual information from which useful conclusions can be derived. Our kinship axioms are also definitions; they have the form **∀ x,y P(x,y) ⇔…** The axioms define the **Mother function and the Husband, Male, Parent, Grandparent, and Sibling predicates** in terms of other predicates.

Some are theorems—that is, they are entailed by the axioms. For example, consider the **assertion that siblinghood is symmetric: ∀ x,y Sibling(x,y) ⇔ Sibling(y,x)** .

Numbers

NatNUM

We describe here the theory of natural numbers or non-negative integers.Natural numbers are defined recursively

- 0 is a natural number : **NatNum(0)** .
- For every object n, if n is a natural number, then S(n) is a natural number :

  $\forall$ n NatNum(n) $\Rightarrow$ NatNum(S(n))

So the natural numbers are 0, S(0), S(S(0)), and so on.

Axioms

$\forall$ n, 0 $\neq$ S(n) .

$\forall$ m,n m $\neq$ n $\Rightarrow$ S(m) $\neq$ S(n) .

Note : We can also write S(n) as n + 1

Numbers

Definition : Addition is defined in terms of the successor function:

- $\forall m\ NatNum(m) \Rightarrow +(0,m) = m$.
- $\forall m,n\ NatNum(m) \wedge NatNum(n) \Rightarrow +(S(m),n) = S(+(m,n))$

  or
- $\forall m,n\ NatNum(m) \wedge NatNum(n) \Rightarrow (m + 1) + n = (m + n) + 1$

Note : The use of infix notation(like m+1,m+n,etc) is an example of **syntactic sugar,** that is, an extension to or abbreviation of the standard syntax that does not change the semantics.

Sets

- The domain of sets is also fundamental to mathematics as well as to commonsense reasoning. We will use the normal vocabulary of set theory as syntactic sugar.
- The **empty set** is a constant written as **{ }**.
- There is one **unary predicate**, **Set**, which is true of sets.
- The **binary predicates** are **x∈ s** (x is a member of set s) and **s1 ⊆ s2** (set **s1** is a subset, not necessarily proper, of set **s2**).
- The binary functions are
  - **s1** ∩ **s2** (the intersection of two sets),
  - **s1** ∪ **s2** (the union of two sets), and
  - **{x|s}** (the set resulting from adjoining element x to set s).

One possible set of axioms of Sets is as follows:

1. The only sets are the empty set and those made by adjoining something to a set:
   - **$\forall$ s Set(s) $\Leftrightarrow$ (s = { }) $\vee$ ($\exists$ x,s2 Set(s2) $\wedge$ s = {x|s2})**
2. The empty set has no elements adjoined into it. In other words, there is no way to decompose { } into a smaller set and an element:
   - **$\neg\exists$ x,s {x|s} = { }** .
3. Adjoining an element already in the set has no effect:
   - **$\forall$ x,s x$\in$ s $\Leftrightarrow$ s = {x|s} .**
4. The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s2 adjoined with some element y, where either y is the same as x or x is a member of s2:
   - **$\forall$ x,s x$\in$ s $\Leftrightarrow$ $\exists$ y,s2 (s = {y|s2} $\wedge$ (x = y $\vee$ x$\in$ s2)) .**

One possible set of axioms of Sets is as follows:

5. A set is a subset of another set if and only if all of the first set's members are members of the second set:
   - **$\forall$ s1,s2 s1 $\subseteq$ s2 $\Leftrightarrow$ ($\forall$ x x$\in$ s1 $\Rightarrow$ x$\in$ s2) .**

6. Two sets are equal if and only if each is a subset of the other:
   - **$\forall$ s1,s2 (s1 = s2) $\Leftrightarrow$ (s1 $\subseteq$ s2 $\wedge$ s2 $\subseteq$ s1) .**

7. An object is in the **intersection** of two sets if and only if it is a member of both sets:
   - **$\forall$ x,s1,s2 x$\in$ (s1 $\cap$ s2) $\Leftrightarrow$ (x$\in$ s1 $\wedge$ x$\in$s2) .**

8. An object is in the union of two sets if and only if it is a member of either set:
   - **$\forall$ x,s1,s2 x$\in$ (s1 $\cup$ s2) $\Leftrightarrow$ (x$\in$ s1 $\vee$ x$\in$s2) .**

Lists

- Lists are similar to sets. The differences are that lists are *ordered and the same element can appear more than once in a list*.
- **Nil** is the constant list with no elements;
- **Cons, Append, First, and Rest** are functions; and
- **Find** is the predicate that does for lists what Member does for sets.
- **List?** is a predicate that is true only of lists.
- The empty list is [].
- The term **Cons(x,y)**, where y is a nonempty list, is written **[x|y]**.
- The term **Cons(x, Nil)** (i.e., the list containing the element x) is written as **[x]**.
- A list of several elements, such as **[A,B,C]**, corresponds to the nested term **Cons(A, Cons(B, Cons(C, Nil))).**

Wumpus World

- The wumpus agent receives a percept vector with five elements. 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:
  - **∀ t,s,g,m,c Percept([s, Breeze,g,m,c],t) ⇒ Breeze(t) ,**
  - **∀ t,s,b,m,c Percept([s,b, Glitter ,m,c],t) ⇒ Glitter (t) ,**

and so on. These rules exhibit a trivial form of the reasoning process called perception,

Wumpus World

- Simple “reflex” behavior can also be implemented by quantified implication sentences.
  - For example, we have **∀ t Glitter (t) ⇒ BestAction(Grab,t)** .

- Adjacency of any two squares can be defined as

  ∀ x,y,a,b Adjacent([x,y], [a,b]) ⇔

  (x = a ∧ (y = b − 1 ∨ y = b + 1)) ∨ (y = b ∧ (x = a − 1 ∨ x = a + 1))

- We can then say that objects can only be at one location at a time:

  ∀ x,s1,s2,t At(x,s1,t) ∧ At(x,s2,t) ⇒ s1 = s2

- Given its current location, the agent can infer properties of the square from properties of its current percept. 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) .

Wumpus World

- The agent can deduce where the pits are (and where the wumpus is)

  $\forall s\ Breezy(s) \Leftrightarrow \exists r\ Adjacent(r,s) \wedge Pit(r)$.
- **Axiom :**

  $\forall t\ HaveArrow(t + 1) \Leftrightarrow (HaveArrow(t) \wedge \neg Action(Shoot,t))$.

Knowledge Engineering in First Order Logic

- The overall process of constructing a knowledge base, known as **knowledge engineering**
- Knowledge engineering projects can vary in many ways, but they all generally include the following steps
  1. **Identify the Task**
  2. **Assemble Relevant Knowledge**
  3. **Decide on Vocabulary**
  4. **Encode General Knowledge**
  5. **Encode Specific Problem Instances**
  6. **Pose Queries and Get Answers**
  7. **Debug the Knowledge Base**

The Electronic Circuits Domain

In this section, we will create an ontology and knowledge base to help us understand digital circuits, following the seven steps of knowledge engineering.

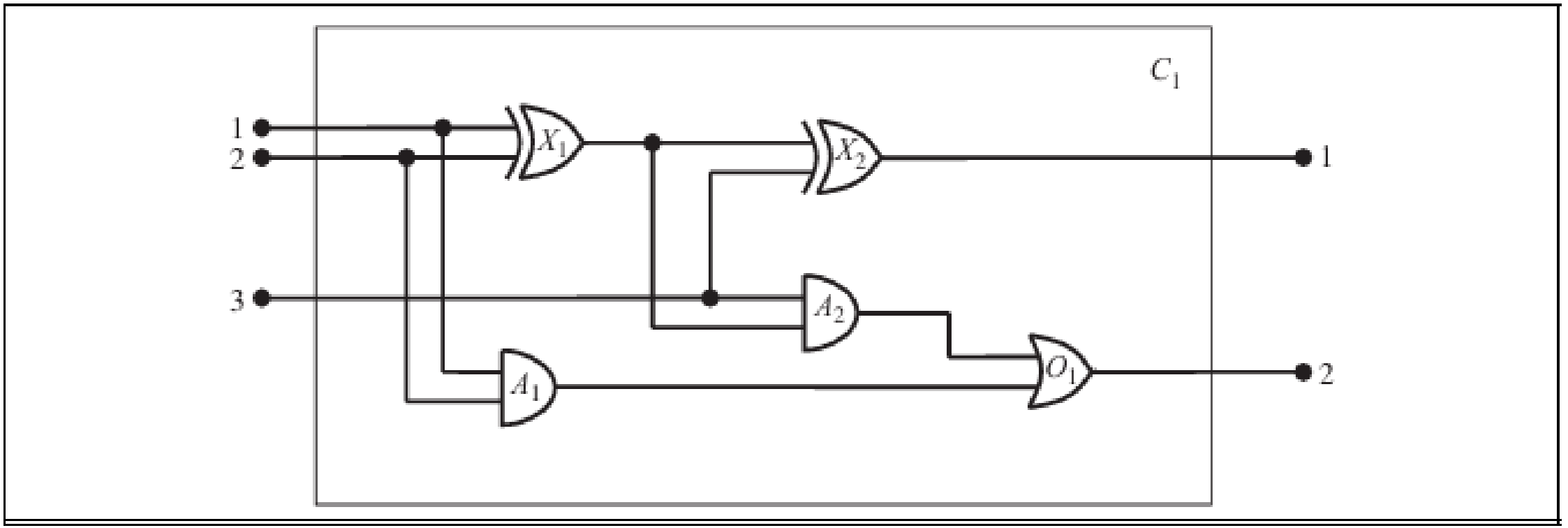

Fig : A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two ANDgates, and one OR gate.

1. Identify the Task

Our main tasks include:

- Checking if the circuit adds correctly.
- Finding out what the output of gate A2 is when all inputs are high.
- Identifying which gates are connected to the first input terminal.
- Looking for feedback loops in the circuit. There are also more detailed analyses involving timing delays, circuit area, power consumption, and production cost. Each of these requires additional knowledge.

2. Assemble Relevant Knowledge

Digital circuits consist of wires and gates.

- Signals travel along wires to the **input terminals of gates**, which then produce **output signals on another terminal**.
- Gates can be of four types: **AND, OR, XOR** (each with two inputs), and **NOT** (with one input).
- We focus on the **connections between terminals**, not the wires themselves. **For our analysis, the size, shape, and cost of components are not relevant**.
- If we were debugging **faulty circuits**, we would need to consider wires, since a faulty wire can affect the signals.

3. Decide on Vocabulary

- **Gates**: Each gate is represented as an object with a name, e.g., **Gate(X1),** and its type is defined using a function, like **Type(X1)=XOR**.
- **Circuits**: Represented by a predicate, e.g., **Circuit(C1).**
- **Terminals**: Identified using a predicate, e.g., **Terminal(x)**. Each gate can have input and output terminals, denoted by functions **In(1,X1)** and **Out(1,X1)**.
- **Connectivity**: Represented by a predicate Connected, which connects terminals, e.g., **Connected(Out(1,X1),In(1,X2)).**
- **Signal Values**: We can use a predicate **On(t)** to check if a signal is on, but it's easier to use objects **1** and **0** with a function **Signal(t)** to represent signal values.

4. Encode General Knowledge of the Domain

1. **If two terminals are connected, they have the same signal:**
   $$\forall t_1, t_2 \;\; Terminal(t_1) \wedge Terminal(t_2) \wedge Connected(t_1, t_2) \;\Rightarrow$$
   $$Signal(t_1) = Signal(t_2) \; .$$
2. **Each terminal's signal is either 1 or 0**
   $$\forall t \;\; Terminal(t) \;\Rightarrow\; Signal(t) = 1 \vee Signal(t) = 0$$
3. **Connectivity is commutative:**
   $$\forall t_1, t_2 \;\; Connected(t_1, t_2) \;\Leftrightarrow\; Connected(t_2, t_1)$$
4. **There are four types of gates:**
   $$\forall g \;\; Gate(g) \wedge k = Type(g) \;\Rightarrow\; k = AND \vee k = OR \vee k = XOR \vee k = NOT$$

5. **An AND gate outputs 0 if any input is 0:**

$$\forall g \;\; Gate(g) \wedge Type(g) = AND \;\Rightarrow$$
$$Signal(Out(1,g)) = 0 \;\Leftrightarrow\; \exists n \;\; Signal(In(n,g)) = 0$$

6. **An OR gate outputs 1 if any input is 1:**

$$\forall g \;\; Gate(g) \wedge Type(g) = OR \;\Rightarrow$$
$$Signal(Out(1,g)) = 1 \;\Leftrightarrow\; \exists n \;\; Signal(In(n,g)) = 1 \,.$$

7. **An XOR gate outputs 1 if inputs are different:**

$$\forall g \;\; Gate(g) \wedge Type(g) = XOR \;\Rightarrow$$
$$Signal(Out(1,g)) = 1 \;\Leftrightarrow\; Signal(In(1,g)) \neq Signal(In(2,g)) \,.$$

8. **A NOT gate's output is different from its input:**

$$\forall g \;\; Gate(g) \wedge (Type(g) = NOT) \;\Rightarrow\; Signal(Out(1, g)) \neq Signal(In(1, g))$$

9. **AND, OR, and XOR gates have two inputs and one output; NOT gates have one input and one output.**

$$\forall g \;\; Gate(g) \wedge Type(g) = NOT \;\Rightarrow\; Arity(g, 1, 1) \;\; .$$

$$\forall g \;\; Gate(g) \wedge k = Type(g) \wedge (k = AND \vee k = OR \vee k = XOR) \;\Rightarrow\; Arity(g, 2, 1)$$

10. **A circuit has terminals up to its input and output capacity:**

$$\forall c, i, j \;\; Circuit(c) \wedge Arity(c, i, j) \Rightarrow$$
$$\forall n \;\; (n \leq i \Rightarrow Terminal(In(c, n))) \wedge (n > i \Rightarrow In(c, n) = Nothing) \wedge$$
$$\forall n \;\; (n \leq j \Rightarrow Terminal(Out(c, n))) \wedge (n > j \Rightarrow Out(c, n) = Nothing)$$

11. **Gates, terminals, signals, and gate types are all distinct.**

$$\forall g, t \;\; Gate(g) \wedge Terminal(t) \Rightarrow$$
$$g \neq t \neq 1 \neq 0 \neq OR \neq AND \neq XOR \neq NOT \neq Nothing \; .$$

12. **All gates are also circuits:**

$$\forall g \;\; Gate(g) \Rightarrow Circuit(g)$$

5.Encode the Specific Problem Instance

Circuit: `Circuit(C1) ∧ Arity(C1,3,2)`

Gates:

- `Gate(X1) ∧ Type(X1)=XOR`
- `Gate(X2) ∧ Type(X2)=XOR`
- `Gate(A1) ∧ Type(A1)=AND`
- `Gate(A2) ∧ Type(A2)=AND`
- `Gate(O1) ∧ Type(O1)=OR`

Connections:

$$Connected(Out(1,X_1),In(1,X_2)) \quad Connected(In(1,C_1),In(1,X_1))$$
$$Connected(Out(1,X_1),In(2,A_2)) \quad Connected(In(1,C_1),In(1,A_1))$$
$$Connected(Out(1,A_2),In(1,O_1)) \quad Connected(In(2,C_1),In(2,X_1))$$
$$Connected(Out(1,A_1),In(2,O_1)) \quad Connected(In(2,C_1),In(2,A_1))$$
$$Connected(Out(1,X_2),Out(1,C_1)) \quad Connected(In(3,C_1),In(2,X_2))$$
$$Connected(Out(1,O_1),Out(2,C_1)) \quad Connected(In(3,C_1),In(1,A_2)) .$$

6. Pose Queries to the Inference Procedure

What combinations of inputs would cause the first output of C_1 (the sum bit) to be 0 and the second output of C_1 (the carry bit) to be 1?

$$\exists i_1, i_2, i_3 \;\; Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, C_1)) = i_2 \wedge Signal(In(3, C_1)) = i_3$$
$$\wedge \; Signal(Out(1, C_1)) = 0 \wedge Signal(Out(2, C_1)) = 1 \; .$$

The answers are substitutions for the variables i_1, i_2, and i_3 such that the resulting sentence is entailed by the knowledge base. ASKVARS will give us three such substitutions:

$$\{i_1/1, \; i_2/1, \; i_3/0\} \quad \{i_1/1, \; i_2/0, \; i_3/1\} \quad \{i_1/0, \; i_2/1, \; i_3/1\} \; .$$

What are the possible sets of values of all the terminals for the adder circuit?

$$\exists i_1, i_2, i_3, o_1, o_2 \;\; Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, C_1)) = i_2$$
$$\wedge \; Signal(In(3, C_1)) = i_3 \wedge Signal(Out(1, C_1)) = o_1 \wedge Signal(Out(2, C_1)) = o_2 \; .$$

7. Debug the Knowledge Base

- For example, suppose we forget to assert that 1≠ 0. Suddenly, the system will be unable to prove any outputs for the circuit, except for the input cases 000 and 110.

$$\exists\, i_1, i_2, o \;\; Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, C_1)) = i_2 \wedge Signal(Out(1, X_1)) \,,$$

which reveals that no outputs are known at X_1 for the input cases 10 and 01. Then, we look at the axiom for XOR gates, as applied to X_1:

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; Signal(In(1, X_1)) \neq Signal(In(2, X_1)) \,.$$

If the inputs are known to be, say, 1 and 0, then this reduces to

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; 1 \neq 0 \,.$$

Now the problem is apparent: the system is unable to infer that $Signal(Out(1, X_1)) = 1$, so we need to tell it that $1 \neq 0$.

Contents

1. First Order Logic:
    a. **Representation Revisited,**
    b. **Syntax and Semantics of First Order Logic,**
    c. **Using First Order Logic**.
2. Inference in First Order Logic:
    a. **Propositional Versus First Order Inference,**
    b. **Unification,**
    c. **Forward Chaining,**

Module 4: **Chapter 2**

Inference in First Order Logic:

a. Propositional Versus First Order Inference,
b. Unification,
c. Forward Chaining,

Propositional Versus First Order Inference,

1. Inference rules for quantifiers
2. Reduction to propositional inference

1. Inference rules for quantifiers

- Consider axiom stating that all greedy kings are evil:
  - **∀ x King(x) ∧ Greedy(x) ⇒ Evil(x)** .
- Then it seems quite permissible to infer any of the following sentences:
  - **King(John) ∧ Greedy(John) ⇒ Evil(John)**
  - **King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)**
  - **King(Father (John)) ∧ Greedy(Father (John)) ⇒ Evil(Father (John)) .**
  - ----------

a) The rule of Universal Instantiation (UI for short)

- The rule of **Universal Instantiation (UI for short)** says that we can infer any sentence obtained by **substituting a ground term** (a term without variables) for the variable.
- To write out the inference rule formally, we use **SUBST(θ, α)** denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v \ \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

b) The rule for Existential Instantiation

- In the rule for Existential Instantiation, the variable is replaced by a single new constant symbol. The formal statement is as follows: for any sentence **α**, variable **v,** and constant symbol **k** that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \;\; \alpha}{\text{SUBST}(\{v/k\}, \alpha)} .$$

For example, from the sentence

$$\exists x \;\; Crown(x) \wedge OnHead(x, John)$$

we can infer the sentence

$$Crown(C_1) \wedge OnHead(C_1, John)$$

as long as C_1 does not appear elsewhere in the knowledge base.

2. Reduction to propositional inference (Propositionalization)

- The existentially quantified sentence can be replaced by one instantiation, and universally quantified sentence can be replaced by the set of all possible instantiations.
- For example, suppose our knowledge base contains just the sentences

FOL Inference

$$\forall x \;\; King(x) \wedge Greedy(x) \;\Rightarrow\; Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John)\ .$$

Propositional logic Inference

$$King(John) \wedge Greedy(John) \;\Rightarrow\; Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \;\Rightarrow\; Evil(Richard)\ ,$$

Modus Ponens

- Modus Ponens, also known as **"the law of detachment"**, is a fundamental rule of logic in propositional reasoning. It states that:
- **If a conditional statement is true (e.g., "If P, then Q") and the antecedent (P) is true, then the consequent (Q) must also be true.**
- **Formal Representation**
  - **Premise 1**: P→Q (If P then Q)
  - **Premise 2**: P (P is true)
  - **Conclusion**: Q (Therefore, Q is true)

Example

- **Premise 1**: If it rains, the ground will be wet. ($P \rightarrow Q$)
- **Premise 2**: It rains. (P)
- **Conclusion**: The ground will be wet. (Q)

Unification

- **Generalized Modus Ponens is a lifted version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic.**
- **Generalized Modus Ponens**: For atomic sentences pi , pi ′ , and q, where there is a substitution θ

such that $\text{SUBST}(\theta, p_i') = \text{SUBST}(\theta, p_i)$, for all i,

$$\frac{p_1', \quad p_2', \quad \ldots, \quad p_n', \quad (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)} .$$

There are $n+1$ premises to this rule: the n atomic sentences p_i' and the one implication. The conclusion is the result of applying the substitution θ to the consequent q. For our example:

p_1' is $King(John)$	p_1 is $King(x)$
p_2' is $Greedy(y)$	p_2 is $Greedy(x)$
θ is $\{x/John, y/John\}$	q is $Evil(x)$

$\text{SUBST}(\theta, q)$ is $Evil(John)$.

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:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) .$$

Example :
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 .

Unification

- In first-order logic**, unification is** a process used to find a common instantiation **for two predicates or terms such that they become identical**.

- **A substitution**, on the other hand, is a **mapping of variables to terms**.

Unification Algorithm

function UNIFY(x, y, θ) **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
    θ, the substitution built up so far (optional, defaults to empty)

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

---

function UNIFY-VAR(var, x, θ) **returns** a substitution

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

Unification is the process of finding a substitution that makes two logical expressions identical. The algorithm takes two expressions, x and y, and attempts to find a substitution (θ) that makes them identical.
Here's a breakdown of how the algorithm works:

Base case: If the substitution **θ i**s already marked as a failure, then it returns failure immediately.
Identity check: If **x** and **y** are identical, it means no further unification is needed, and the current substitution **θ** can be returned.
Variable check: If **x** is a variable, it calls the **UNIFY-VAR** function with **x** as the variable and **y** as the expression. If **y** is a variable, it calls **UNIFY-VAR** with y as the variable and x as the expression.
Compound expression check: If both **x** and **y** are compound expressions, it recursively calls **UNIFY** on their arguments and operators.
List check: If both **x** and **y** are lists, it recursively calls UNIFY on their first elements and their remaining elements.
Failure case: If none of the above conditions are met, it returns failure, indicating that x and y cannot be unified.
The UNIFY-VAR function is used when one of the expressions **(x or y)** is a variable. It attempts to create a substitution based on the variable and the expression it's being unified with.

The UNIFY-VAR function is used when one of the expressions (x or y) is a variable. It attempts to create a substitution based on the variable and the expression it's being unified with.

Substitution check: If the substitution already contains a mapping for the variable, it recursively calls UNIFY with the mapped value and the expression x.

Reverse substitution check: If the expression is already in the substitution, it recursively calls UNIFY with the variable and the mapped value.

Occur check: Checks for a possible occurrence of the variable in the expression, preventing infinite loops, and returns failure if such an occurrence is detected.

Substitution addition: If none of the above cases apply, it adds a new mapping to the substitution, indicating that the variable is unified with the expression.

Overall, the algorithm systematically traverses through the expressions, handling variables, compounds, lists, and checking for failures, until it either finds a successful substitution or determines that unification is not possible.

- Overall, the algorithm systematically traverses through the
  - **expressions,**
  - **handling variables**,
  - **Compound statements**,
  - **lists**, and
  - **checking for failures**,
- until it either finds a ***successful substitution*** or determines that unification is ***not possible***.

Example

Suppose we have the following two predicates:

1. Predicate **P(x,y)**
2. Predicate **Q(f(z),a)**

Here,

- **P** and **Q** are predicates,
- **x, y, and z** are variables, and
- **f and a** are constants.

Now, let's say we want to unify **P(x,y) with Q(f(z),a).**

We can use the given algorithm for unification to find a substitution that makes these two predicates identical.

1. Initially, θ is empty.

2. Start unifying the predicates: P(x,y) and Q(f(z),a)

Since **P** and **Q** are different, they can't be unified directly.

3. Unify the arguments: Unify **x** with **f(z)** and **y** with **a**

4. **Unify x with f(z):**
    - **x** is a variable, **f(z)** is a compound term.
    - Call **UNIFY-VAR(x, f(z), θ):**
        - Add **x/f(z) to θ**
        - **θ={x/f(z)}**
5. **Unify y with a:**
    - **y** is a variable, **a** is a constant.
    - Call UNIFY-VAR(y, a, θ):
        - Add **y/a to θ**
        - **θ={x/f(z),y/a}**

6.Finally, return θ:

θ={x/f(z),y/a}

So, the resulting substitution θ makes P(x,y) and Q(f(z),a) identical:

P(x,y){x/f(z),y/a}=Q(f(z),a)

Storage and retrieval

- Underlying the **TELL** and **ASK** functions used to inform and interrogate a knowledge base are the more primitive **STORE** and **FETCH** functions. STORE(s) stores a sentence s into the knowledge base and **FETCH(q)** returns all unifiers such that the query **q** unifies with some.
- Given a sentence to be stored, it is possible to construct indices for all possible queries that unify with it. For the fact **Employs(IBM , Richard),** the queries are
- **Employs(IBM , Richard)** **Does IBM employ Richard?**
- **Employs(x, Richard)** **Who employs Richard?**
- **Employs(IBM , y)** **Whom does IBM employ?**
- **Employs(x, y)** **Who employs whom?**

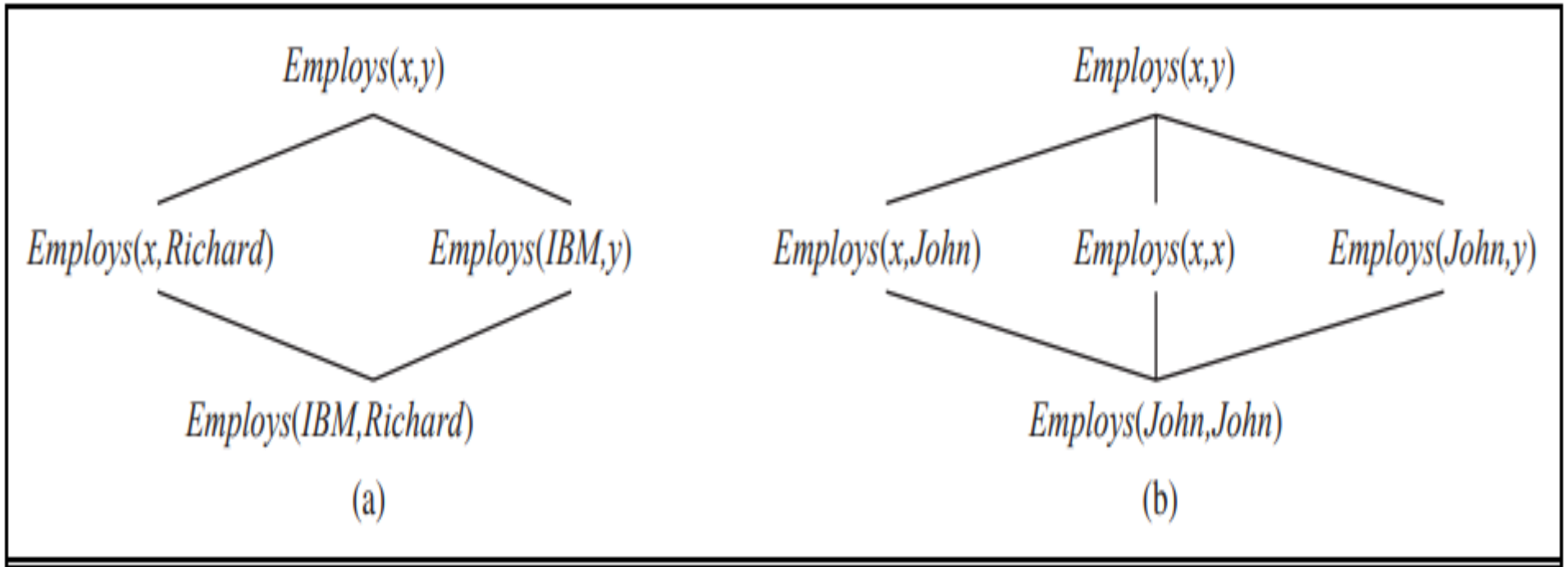

Figure 9.2 (a) The subsumption lattice whose lowest node is *Employs*(*IBM*, *Richard*). (b) The subsumption lattice for the sentence *Employs*(*John*, *John*).

Forward Chaining:

- Forward chaining is a reasoning method , starts with the **known facts and uses inference rules to derive new** conclusions until the goal is reached or no further inferences can be made.
- In essence, it proceeds forward **from the premises to the conclusion.**

Example : Consider the following knowledge base representing a simple diagnostic system:

1.If a patient has a fever, it might be a cold.

2.If a patient has a sore throat, it might be strep throat.

3.If a patient has a fever and a sore throat, they should see a doctor.

Given the facts:

- **The patient has a fever.**
- **The patient has a sore throat.**

- **Forward chaining would proceed as follows:**

1.Check the first rule: **Fever**? **Yes. Proceed**.

2.Check the second rule: **Sore throat**? **Yes. Proceed**.

3.Apply the third rule: The patient has a fever and sore throat, thus they should see a doctor.

Forward chaining is suitable for situations where there is a large amount of known information and the goal is to derive conclusions.

Forward Chaining,

- Start with the atomic sentences in the knowledge base and apply **Modus Ponens** in the forward direction, adding new atomic sentences, until no further inferences can be made.
- **First-order definite clauses** : A definite clause either is **atomic** or is an **implication** whose antecedent is a **conjunction of positive literals** and whose consequent is a **single positive literal**. The following are first-order definite clauses:
  - **King(x) ∧ Greedy(x) ⇒ Evil(x) .**
  - **King(John) .**
  - **Greedy(y)** .

Forward Chaining,

- Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified.
- **Consider the following problem:** ***The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.***
- We will prove that **West** is a criminal.

First, we will represent these facts as first-order definite clauses.

1. **". . . it is a crime for an American to sell weapons to hostile nations":**
   - **American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal(x) .**
2. **"Nono . . . has some missiles."**
   - **The sentence ∃ x Owns(Nono, x)∧Missile(x) is transformed into two definite clauses by Existential Instantiation, introducing a new constant M1:**
     - **Owns(Nono, M1)**
     - **Missile (M1)**
3. **"All of its missiles were sold to it by Colonel West":**
   - **Missile(x) ∧ Owns(Nono, x) ⇒ Sells(West, x, Nono) .**
4. **We will also need to know that missiles are weapons:**
   - **Missile(x) ⇒ Weapon(x)**
5. **and we must know that an enemy of America counts as "hostile":**
   - **Enemy(x, America) ⇒ Hostile(x) .**
6. **"West, who is American . . .":**
   - **American(West) .**
7. **"The country Nono, an enemy of America . . .":**
   - **Enemy(Nono, America) .**

From these inferred facts, we can conclude that Colonel West is indeed a criminal since he sold missiles to a hostile nation, which is Nono.

“. . . it is a crime for an American to sell weapons to hostile nations”:

- **American(West) ∧ Weapon(Missile) ∧ Sells(West, Missile, Nono) ∧ Hostile(Nono) ⇒ Criminal(West) .**

DATALOG

- This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases.
- **Datalog** is a language that is restricted to first-order definite clauses with no function symbols.
- **Datalog** gets its name because it can represent the type of statements typically made in relational databases.

Forward Chaining Algorithm: Example

Knowledge Base (`KB`):

1. `Parent(John, Mary)`
2. `Parent(Mary, Alice)`
3. `Parent(x, y) ∧ Parent(y, z) ⇒ Grandparent(x, z)`

Query (`α`):

`Grandparent(John, Alice)`

1. **Iteration 1:**
    - Rule: `Parent(x, y) ∧ Parent(y, z) ⇒ Grandparent(x, z)`
    - Possible substitutions:
        - For `x=John`, `y=Mary` from `Parent(John, Mary)`.
        - For `y=Mary`, `z=Alice` from `Parent(Mary, Alice)`.
        - Combined substitution `θ = {x=John, y=Mary, z=Alice}` satisfies the premise.
    - Conclusion: Apply `θ` to `Grandparent(x, z)` → `Grandparent(John, Alice)`.
    - Add `Grandparent(John, Alice)` to `new`.
2. **Unification:**
    - `Grandparent(John, Alice)` unifies with `α`.
    - Return substitution `φ = {x=John, z=Alice}`.

Explanation of Algorithm

- This algorithm is an implementation of Forward Chaining with a **goal-directed query mechanism, specifically designed for First-Order Logic (FOL) knowledge bases**.
- It's called Forward Chaining with Ask (FOL-FC-ASK). Let's break down the steps:

Algorithm:

1. **Inputs:**
   - **KB**: The knowledge base, which consists of a set of first-order definite clauses.
   - **α:** The query, which is an atomic sentence.
2. **Loop until no new sentences are inferred:**
   - Initialize **new** as an empty set.
3. **Iterate through each rule in the knowledge base**:
   - Standardize the variables in the rule (**ensuring variable names are unique**).
   - For each **substitution θ** that makes the antecedent of the rule (`p1 ∧ ... ∧ pn`) match some subset of the KB:
     1. Apply the substitution to the consequent of the rule (`q`) to generate a new sentence `**q'**`.
     2. Check if `**q'**` unifies with some sentence already in the KB or `new`. If not, add `q'` to `new`.
     3. Attempt to unify `**q'**` with the query `**α**`. If unification succeeds (resulting in a substitution ϕ), return ϕ.
     4. **Update the knowledge base**:
        - Add the sentences in `new` to the KB
     5. **Repeat the loop** until no new sentences are inferred or until the query is proven or disproven.
4. **Output**:

•If the query is proven, return the substitution that makes it true.

•If the query is disproven (i.e., it cannot be proven true), return false.

A simple forward-chaining algorithm

function FOL-FC-ASK(KB, α) **returns** a substitution or *false*
  inputs: KB, the knowledge base, a set of first-order definite clauses
    α, the query, an atomic sentence
  local variables: *new*, the new sentences inferred on each iteration

  repeat until *new* is empty
    new $\leftarrow \{\}$
    for each *rule* **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(*rule*)
      for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
          for some $p'_1, \ldots, p'_n$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' does not unify with some sentence already in KB or *new* **then**
          add q' to *new*
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *fail* **then return** ϕ
    add *new* to KB
  return *false*

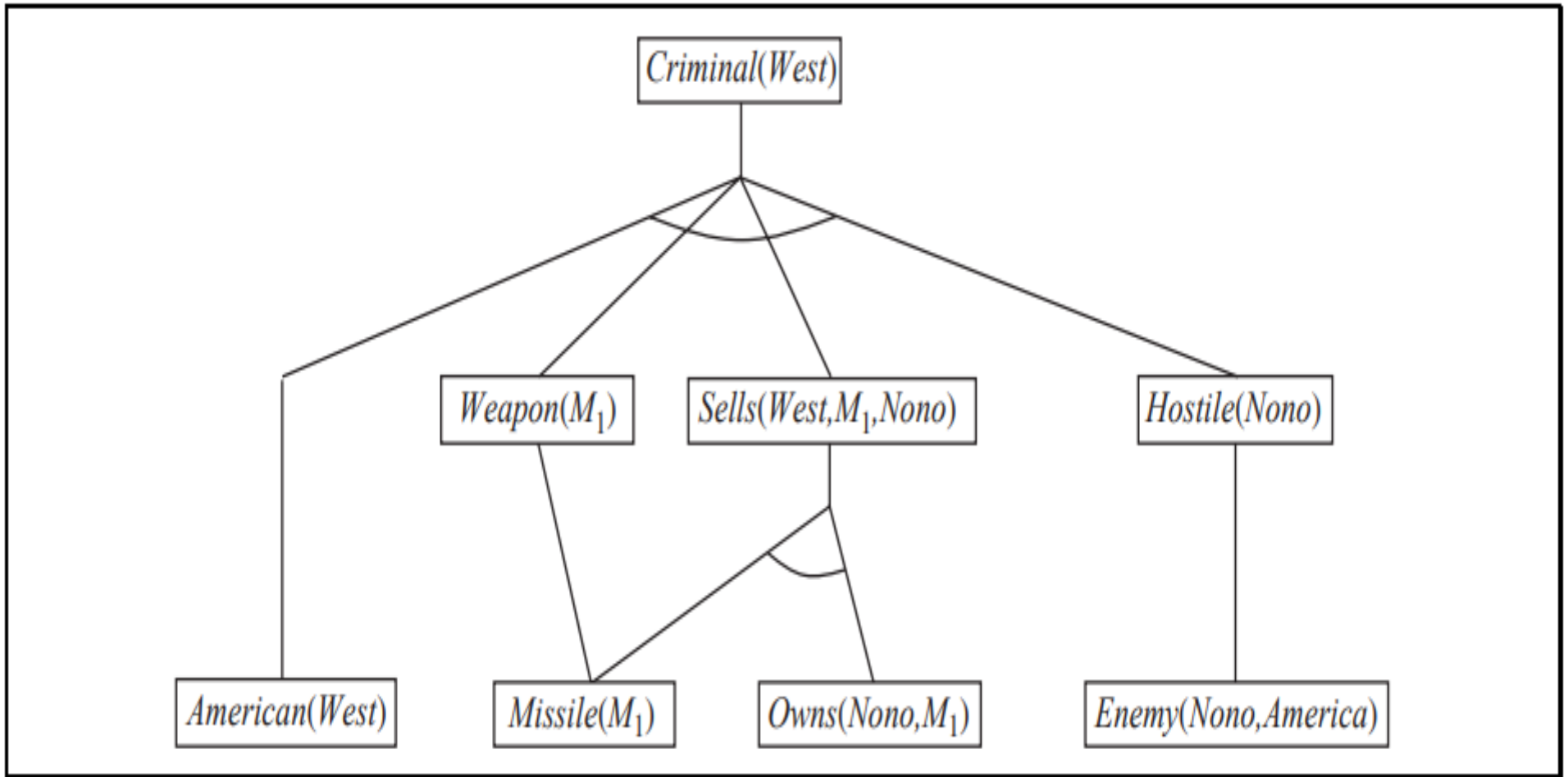

Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Summary

1. F**orward chaining starts** with known facts and moves forward to reach conclusions,
2. **Backward chaining** starts with the goal and moves backward to verify if the goal can be satisfied, and (Module 5)
3. **Resolution** is an inference rule used to derive new clauses by combining existing ones. (Module 5)

These ***techniques are essential for reasoning and inference in First-Order Logic systems***.