(1) What are the steps associated with the knowledge Engineering process?

Discuss them by applying the steps to any real world application of your choice.

**Knowledge Engineering**

The general process of constructing knowledge base is 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. We will illustrate the knowledge engineering process in an electronic circuit domain that should already be fairly familiar,

**The steps associated with the knowledge engineering process are :**

1. **Identify the task.**
   The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents.

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.

3. **Decide on a vocabulary of predicates, functions, and constants.** That is, translate the important domain-level concepts into logic-level names. Once the choices have been made, the result is a vocabulary that is known as the ontology of the domain. The word ontology means a particular theory of the nature of being or existence.

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**
   For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied 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.

7. **Debug the knowledge base.**

\[ \forall x \text{ NumOfLegs}(x,4) \Rightarrow \text{Mammal}(x) \] is false for reptiles, amphibians.

To understand this seven-step process better, we now apply it to an extended example—the domain of electronic circuits.

**The electronic circuits domain**

| One | bit | adder |
1 Identify the task

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks in this section.

2 Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire.

3 Decide on a vocabulary

We now know that we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them. We will start from individual gates and move up to circuits.

First, we need to be able to distinguish a gate from other gates. This is handled by naming gates with constants: $X_1$, $X_2$, and so on

\[
\text{Type}(X_1) = \text{XOR} \\
\text{Type}(X_1, \text{XOR}) - \text{binary predicate} \\
\text{XOR}(X_1) - \text{individual type}
\]

A gate or circuit can have one or more terminal. For $X_1$ the terminals are $X_1\text{In}_1$, $X_1\text{In}_2$, $X_1\text{Out}_1$

\[
X_1\text{In}_1, 1^{st} \text{ input of gate } X_1 \\
X_1\text{In}_2, 2^{nd} \text{ input of gate } X_1 \\
X_1\text{Out}_1, \text{Output of gate } X_1
\]

4 Encode general knowledge of the domain

One sign that we have a good ontology is that there are very few general rules which need to be specified. A sign that we have a good vocabulary is that each rule can be stated clearly and concisely. With our example, we need only seven simple rules to describe everything we need to know about circuits:

1. If two terminals are connected, then they have the same signal:

\[
\forall t_1, t_2 \text{ Connected}(t_1, t_2) \Rightarrow \text{Signal}(t_1) = \text{Signal}(t_2)
\]
2. The signal at every terminal is either 1 or 0 (but not both):
   \[ \forall t \text{ Signal}(t) = 1 \lor \text{Signal}(t) = 0 \]
   \[ 1 \neq 0 \]

3. Connected is a commutative predicate:
   \[ \forall t_1, t_2 \text{ Connected}(t_1, t_2) \Rightarrow \text{Connected}(t_2, t_1) \]

4. An OR gate's output is 1 if and only if any of its inputs is 1:
   \[ \forall g \text{ Type}(g) = \text{OR} \Rightarrow \text{Signal}(\text{Out}(1,g)) = 1 \iff \exists n \text{ Signal}(\text{In}(n,g)) = 1 \]

5. An AND gate's output is 0 if and only if any of its inputs is 0:
   \[ \forall g \text{ Type}(g) = \text{AND} \Rightarrow \text{Signal}(\text{Out}(1,g)) = 0 \iff \exists n \text{ Signal}(\text{In}(n,g)) = 0 \]

6. An XOR gate's output is 1 if and only if its inputs are different:
   \[ \forall g \text{ Type}(g) = \text{XOR} \Rightarrow \text{Signal}(\text{Out}(1,g)) = 1 \iff \text{Signal}(\text{In}(1,g)) \neq \text{Signal}(\text{In}(2,g)) \]

7. A NOT gate's output is different from its input:
   \[ \forall g \text{ Type}(g) = \text{NOT} \Rightarrow \text{Signal}(\text{Out}(1,g)) \neq \text{Signal}(\text{In}(1,g)) \]

5 Encode the specific problem instance
The circuit shown in Figure is encoded as circuit \( C_1 \) with the following description. First, we categorize the gates:

\[ \text{Type}(X_1) = \text{XOR} \quad \text{Type}(X_2) = \text{XOR} \]
\[ \text{Type}(A_1) = \text{AND} \quad \text{Type}(A_2) = \text{AND} \]
\[ \text{Type}(O_1) = \text{OR} \]

\[ \text{Connected}(\text{Out}(1,X_1),\text{In}(1,X_2)) \]
\[ \text{Connected}(\text{Out}(1,X_1),\text{In}(2,A_2)) \]
\[ \text{Connected}(\text{Out}(1,A_1),\text{In}(1,O_1)) \]
\[ \text{Connected}(\text{Out}(1,A_1),\text{In}(2,O_1)) \]
\[ \text{Connected}(\text{Out}(1,X_2),\text{Out}(1,C_1)) \]
\[ \text{Connected}(\text{Out}(1,O_1),\text{Out}(2,C_1)) \]

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?

\[ \exists i_1, i_2, o_1, o_2 \text{ Signal}(\text{In}(1,C_1)) = i_1 \land \text{Signal}(\text{In}(2,C_1)) = i_2 \land \text{Signal}(\text{In}(3,C_1)) = i_3 \land \text{Signal}(\text{Out}(1,C_1)) = 0 \land \text{Signal}(\text{Out}(2,C_1)) = 1 \]
The answer are substitutions for the variable i1,i2 and i3 such that the resulting sentence is entailed by the knowledge base. There are three such substitutions:

\{i1/1,i2/1,i3/0\} \{i1/2,i2/0,i3/1\} \{i1/0,i2/1,i3/1\}

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

\[\exists i_1,i_2,i_3,o_1,o_2 \quad \text{Signal(In(1,C_1))} = i_1 \land \text{Signal(In(2,C_0))} = i_2 \land \text{Signal(In(3,1))} = i_3 \land \text{Signal(Out(1,CO))} = 01 \land \text{Signal(Out(2,CO))} = 02\]

This final query will return a complete input/output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of circuit verification.

7 Debug the knowledge base

The knowledge base is checked with different constraints. For example if the assertion 1#0 is not included in the knowledge base then it is variable to prove any output for the circuit, expect for the input cases 001 and 110.

\[\exists i_1,i_2,o \quad \text{Signal(In(1,C_1))} = i_1 \land \text{Signal(In(2,C_1))} = i_2 \land \text{Signal(Out(1,X_1))}\]

This claim that no output are known at X1 for the input cases 10 and 01. Therefore the axiom for XOR gate is considered.

\[\text{Signal(Out(1,X_1))} = 1 \iff \text{Signal(In(1,X_1))} \neq \text{Signal(In(2,X_1))}\]

If the input signal are known 1 and 0, the above sentence reduces to \(\text{Signal(Out(1,X_1))} = 1 \iff 1 \neq 0\)

Now the system is unable to infer the \(\text{Signal(Out(1,X_1))} = 1\)

Unification
Generalized Modus Pones

- For atomic sentences \(p_i, p_i', q\) where there is a substitution \(q\) such that \(\text{Subst}\{q, p_i'\} = \text{Subst}\{q, p_i\}\):

\[p_1', p_2', \ldots, p_n', (p_1 \land p_2 \land \ldots \land p_n \Rightarrow q) \quad \text{Subst}(\theta, q)\]
Unification

- It is the process of finding substitutions that make two atomic sentences identical
- Lifted inference rules require finding substitution that make different logical expression look identical this process is called unification
- It is the key component of first order inference logic algorithm

\[
\text{UNIFY}(p, q) = \theta \quad \text{where} \quad \text{SUBST}(\theta, p) = \text{SUBST}(q, \theta)
\]

\(\theta\)- Unifier of two sentences

\(P-S1(x, x)\)

\(Q-S1(y, z)\)

Assume \(\theta = y\)

Unification algorithm returns failure as the result at two factors

i) Two sentence have different predicate name
   
   Ex. hate(m,c)
   
   try(m,c)

ii) Two sentence will have different number of arguments
    
    Ex. \(\text{hate}(m, c, x)\)
    
    \(\text{hate}(m, c)\)

**Query Knows (John, x) whom does john knows**

The knowledge base contain the following information

Knows(John, x)

Knows(John, Jane)

Knows(y, Bill)

Knows(y, Mother(y))

Knows(x, Elizabeth)

\[
\text{UNIFY}(\text{Knows}(John, x), \text{Knows}(John, Jane)) = \{x/Jane\}
\]

\[
\text{UNIFY}(\text{Knows}(John, x), \text{Knows}(y, Bill)) = \{x/Bill, y/John\}
\]

\[
\text{UNIFY}(\text{Knows}(John, x), \text{Knows}(y, Mother(y))) = \{y/John, x/Mother(John)\}
\]

\[
\text{UNIFY}(\text{Knows}(John, x), \text{Knows}(x, Elizabeth)) = \text{fail}.
\]

**Consider the last sentence:**
UNIFY\((\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth}))\) = fail

• This fails because \(x\) cannot take on two values
• But “Everyone knows Elizabeth” and it should not fail
• Must standardize apart one of the two sentences to eliminate reuse of variable

UNIFY\((\text{Knows}(\text{John}, x), \text{Knows}(z_{17}, \text{Elizabeth}))\) = \{x/\text{Elizabeth}, z_{17}/\text{John}\}

Standardizing apart eliminates overlap of variables, e.g., \(\text{Knows}(z_{17}, OJ)\) by renaming its variables to avoid name clashes

Most General Unifier

Multiple unifiers are possible:

\[
\begin{align*}
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z)) & = \{y/\text{John}, x/z\} \\
& \quad \text{or} \\
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z)) & = \{y/\text{John}, x/\text{John}, z/\text{John}\}
\end{align*}
\]

Which is better, \(\text{Knows}(\text{John}, z)\) or \(\text{Knows}(\text{John}, \text{John})\)?

• Second could be obtained from first with extra subs
• First unifier is more general than second because it places fewer restrictions on the values of the variables

There is a single most general unifier for every unifiable pair of expressions

Occur Check

\(t_1 = \text{likes}(X, Y)\)
\(t_2 = \text{likes}(g(Y))\)

An equation that has a variable on one side and a term containing that variable on the other side cannot be solved no matter what the substitute is. So the unification fails. The above phenomenon is called "occurs check".

Unification algorithm
Storage and retrieval

Once the data type for sentences and terms are defined, we need to contain a set of sentences in a KB

i) Stores(s) – store a sentence s

ii) Fetch(s)-returns all unifiers

such that query q unifies with some sentence in KB.

An example is when we ask Knows (John, x) – is an instance of fetching

- The simplest way to implement STORE and FETCH is to keep all the facts in the knowledge base in one long list
- The given query UNIFY(q,s) – given query call for every sentence s in the list, requires O(n) time on an n-element KB.
- Such a process is inefficient in terms of time taken because it will unify each and every statement of knowledge base
- Inefficient because we’re performing so many unifies.

function **UNIFY**(x, y, θ) returns a substitution to make x and y identical

inputs: x, a variable, constant, list, or compound

y, a variable, constant, list, or compound

θ, the substitution built up so far

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(ARGS[x], ARGs[y], UNIFY(OP[x], OP[y], θ))
else if LIST?(x) and LIST?(y) then
    return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
else return failure

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

inputs: var, a variable

x, any expression

θ, the substitution built up so far

if {var/val} ∈ θ then return UNIFY(val, x, θ)
else if {x/val} ∈ θ then return UNIFY(var, val, θ)
else if OCCUR-CHECK?(var, x) then return failure
else return add {var/x} to θ
• Example: unify Knows (John, x) with Brother (Richard, John) 
  There is no need to unify
  ➢ We can avoid such unification by indexing the facts in the knowledge base
  ➢ Predicate indexing puts all the Knows facts in one bucket and all the Brother facts in another
  ➢ The bucket stored in a hash table for efficient access.
  ➢ We can large bucket problem with the help of multiple indexing
    • Might not be a win if there are lots of clauses for a particular predicate symbol
      ▪ Consider how many people Know one another

  – Instead index by predicate and first argument
  ▪ Clauses may be stored in multiple buckets

Subsumption lattice
The above set of queries are said to form a collection tree called as Subsumption lattice

How to construct indices for all possible queries that unify with it
• Example: Employs (AIMA.org, Richard)

\[
\begin{align*}
\text{Employs(AIMA.org, Richard)} & \quad \text{Does AIMA.org employ Richard?} \\
\text{Employs(x, Richard)} & \quad \text{Who employs Richard?} \\
\text{Employs(AIMA.org, y)} & \quad \text{Whom does AIMA.org employ?} \\
\text{Employs(x, y)} & \quad \text{Who employs whom?}
\end{align*}
\]

\[
\begin{array}{c}
\text{Employs(x,y)} \\
\text{Employs(x,Richard)} \quad \text{Employs(AIMA.org,y)} \\
\text{Employs(AIMA.org,Richard)} \\
\end{array}
\begin{array}{c}
\text{Employs(x,y)} \\
\text{Employs(x,John)} \quad \text{Employs(x,x)} \quad \text{Employs(John,y)} \\
\text{Employs(John,John)}
\end{array}
\]

\textbf{Figure 9.2} (a) The subsumption lattice whose lowest node is the sentence \text{Employs(AIMA.org, Richard)}. (b) The subsumption lattice for the sentence \text{Employs(John, John)}.

Properties of subsumption lattice
• The child of any node in the lattice is obtained from its parent by single substitution
• The “highest” common descendent of any two nodes is the result of applying the most general unifier
• The portion of lattice , above any ground fact can be construct
• Predicate with n arguments will create a lattice with O(2^n) nodes
• Benefits of indexing may be outweighed by cost of storing and maintaining indices

- Resolution is a procedure used in proving that arguments which are expressible in predicate logic are correct
- Resolution is a procedure that produces proofs by refutation or contradiction
- Resolution lead to refute a theorem-proving technique for sentences in propositional logic and first order logic

- Resolution is a rule of inference
- Resolution is computerized theorem prover
- Resolution is so far only defined for propositional logic. The strategy is that the resolution techniques of propositional logic be adopted in predicate logic

Procedure for resolution

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

A Predicate Logic Example

1. Marcus was a man. \( \text{man(Marcus)} \)
2. Marcus was a Pompeian. \( \text{Pompeian(Marcus)} \)
3. All Pompeians were Romans. \( \forall x: \text{Pompeian}(x) \rightarrow \text{Roman}(x) \)
4. Caesar was a ruler. \( \text{ruler(Caesar)} \)
5. All Romans were either loyal to Caesar or hated him.
\( \forall x: \text{Roman}(x) \rightarrow \text{loyalty}(x, \text{Caesar}) \lor \text{hate}(x, \text{Caesar}) \)
6. Everyone is loyal to someone. \( \forall x: \exists y: \text{loyalty}(x, y) \)
7. People only try to assassinate rulers they aren’t loyal to.
\( \forall x: \exists y: \text{person}(x) \land \text{ruler}(y) \land \text{tryassassinate}(x, y) \rightarrow \neg \text{loyalty}(x, y) \)
8. Marcus tried to assassinate Caesar.
\( \text{tryassassinate(Marcus, Caesar)} \)
9. All men are people. \( \forall x: \text{man}(x) \rightarrow \text{person}(x) \)

Conversion to Clause Form

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

Solution:
- Flatten
- Separate out quantifiers

Conjunctive Normal Form:
\[ \neg \text{Roman}(x) \lor \neg \text{know}(x, \text{Marcus}) \lor \text{hate}(x, \text{Caesar}) \lor \neg \text{hate}(y, z) \lor \text{thinkcrazy}(x, z) \]

Clause Form
- Conjunctive normal form
Converting FOL Sentences to CNF

1. Replace $\leftrightarrow$ with equivalent (added):
   - convert $P \leftrightarrow Q$ to $P \Rightarrow Q \land Q \Rightarrow P$

2. Replace $\Rightarrow$ with equivalent: convert $P \Rightarrow Q$ to $\neg P \lor Q$

3. Reduce scope of $\neg$ to single literals:
   - convert $\neg \neg P$ to $P$ (DNE)
   - convert $\neg (P \lor Q)$ to $\neg P \land \neg Q$ (de Morgan's)
   - convert $\neg (P \land Q)$ to $\neg P \lor \neg Q$ (de Morgan's)
   - convert $\neg \forall x \ P$ to $\exists x \neg P$
   - convert $\neg \exists x \ P$ to $\forall x \neg P$

4. Standardize variables apart:
   - each quantifier must have a unique variable name
   - avoids confusion in steps 5 and 6
   - e.g. convert $\forall x \ P \lor \exists x \ Q$ to $\forall x \ P \lor \exists y \ Q$

5. Eliminate existential quantifiers (Skolemize):
   - convert $\exists x \ P(x)$ to $P(C)$ (EE)
     - $C$ must be a new constant (Skolem constant)
   - convert $\forall x, y \exists x \ P(x, y, z)$ to $\forall x, y \ P(x, y, F(x, y))$
     - $F(\ )$ must be a new function (Skolem function) with arguments that are all enclosing universally quantified variables

**Skolem Functions in FOL**

- **Objective**
  - Want all variables universally quantified
  - Notational variant of FOL w/o existentials
  - Retain implicitly full FOL expressiveness

- **Skolem's Theorem**
  Every existentially quantified variable can be replaced by a unique Skolem function whose arguments are all the universally quantified variables on which the existential depends, without changing FOL.

- **Examples**
  - "Everybody likes something"
    $\forall(x) \exists(y) [Person(x) \land Likes(x, y)]$
  - "Every philosopher writes at least one book"
    $\forall(x) \exists(y)[Philosopher(x) \land Book(y)] \Rightarrow Write(x, y)]$
    $\forall(x)[(Philosopher(x) \land Book(S2(x))) \Rightarrow Write(x, S2(x))]$
6. Drop quantifiers:
   — all variables are only universally quantified after step 5
   — e.g. convert $\forall x \ P(x) \lor \forall y \ Q(y)$ to $P(x) \lor Q(y)$
   — all variables in KB will be assumed to be universally quantified
7. Distribute $\land$ over $\lor$ to get conjunction of disjunctions:
   — convert $(P \land Q) \lor R$ to $(P \lor R) \land (Q \lor R)$

8. Flatten nested conjunctions and disjunctions:
   — convert $(P \land Q) \land R$ to $(P \land Q \land R)$
   — convert $(P \lor Q) \lor R$ to $(P \lor Q \lor R)$

9. Separate each conjunct (added)
   split at $\land$'s so each conjunct is now a CNF clause

   \[ \neg \text{Above}(x, y) \lor \text{OnTop}(x, y) \lor \text{OnTop}(x, F(x, y)) \] \land
   \[ \neg \text{Above}(x, y) \lor \text{OnTop}(x, y) \lor \text{Above}(F(x, y), y) \]

   becomes:
   \[ \neg \text{Above}(x, y) \lor \text{OnTop}(x, y) \lor \text{OnTop}(x, F(x, y)) \]
   \[ \neg \text{Above}(x, y) \lor \text{OnTop}(x, y) \lor \text{Above}(F(x, y), y) \]

10. Standardize variables apart in each clause (added)
    — each clause in KB must contain unique variable names
    — now during unification the standardize apart step
        need only be done on deduced clauses (i.e. resolvents)

    \[ \neg \text{Above}(x, y) \lor \text{OnTop}(x, y) \lor \text{OnTop}(x, F(x, y)) \]
    \[ \neg \text{Above}(x, y) \lor \text{OnTop}(x, y) \lor \text{Above}(F(x, y), y) \]

Examples of Conversion to Clause Form

Example:

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

   • Eliminate $\rightarrow$

\[ \forall x: \neg [\text{Roman}(x) \land \text{know}(x, Marcus)] \lor [\text{hate}(x, Caesar) \lor (\forall y: \]
\[ \exists x: (hate(y,x) \lor thinkcrazy(x,y))] \]

- Reduce scope of \( \neg \).
- Move quantifiers. \( \forall x: \forall y: [\neg Roman(x) \lor know(x, Marcus)] \lor [hate(x, Caesar) \lor \forall y: \forall z: \neg hate(y,z) \lor thinkcrazy(x,y)] \)
- "Standardize" variables:
  \( \forall x: P(x) \lor \forall x: Q(x) \) converts to \( \forall x: P(x) \lor \forall y: Q(y) \)
- Eliminate existential quantifiers.

\[ \exists y: President(y) \lor \text{President}(S1) \]
\[ \forall x: \exists y: father-of(y,x) \Rightarrow \exists y: father-of(S1(x),x) \]
- Drop the prefix.
- Convert to a conjunction of disjuncts.

\[ \neg Roman(x) \lor know(x, Marcus) \lor [hate(x, Caesar) \lor \neg hate(y,z) \lor thinkcrazy(x,y)] \]

**The Basis of Resolution and Herbrand's Theorem**

\[ \text{Given:} \]

<table>
<thead>
<tr>
<th>winter \lor summer</th>
</tr>
</thead>
<tbody>
<tr>
<td>\neg winter \lor cold</td>
</tr>
</tbody>
</table>

We can conclude:

| summer \lor cold |

\[ \text{Herbrand's Theorem:} \]

To show that a set of clauses \( S \) is unsatisfiable, it is necessary to consider only interpretations over a particular set, called the Herbrand universe of \( S \). A set of clauses \( S \) is unsatisfiable if and only if a finite subset of ground instances (in which all bound variables have had a value substituted for them) of \( S \) is unsatisfiable.

**Algorithm: Propositional Resolution**

1. Convert all the propositions of \( F \) to clause form.
2. Negate \( P \) and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
   a) Select two clauses. Call these the parent clauses.
   b) Resolve them together.
   c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.
### A Few Facts in Propositional Logic

<table>
<thead>
<tr>
<th>Given Axioms</th>
<th>Clause Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>$P$</td>
</tr>
<tr>
<td>$(P \land Q) \rightarrow R$</td>
<td>$\neg P \lor \neg Q \lor R$</td>
</tr>
<tr>
<td>$(S \lor T) \rightarrow Q$</td>
<td>$\neg S \lor Q$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

### Resolution in Propositional Logic

\[
\neg P \lor \neg Q \lor R
\]

\[
\neg R
\]

\[
\neg P \lor \neg Q
\]

\[
P
\]

\[
\neg T \lor Q
\]

\[
\neg Q
\]

\[
\neg T
\]

\[
T
\]

\[
\text{Box}
\]
Axioms in clause form:

1. $\text{man}(\text{Marcus})$
2. $\text{Pompeian}(\text{Marcus})$
3. $\neg \text{Pompeian}(x_1) \lor \text{Roman}(x_1)$
4. $\text{Ruler}(\text{Caesar})$
5. $\neg \text{Roman}(x_2) \lor \text{loyalto}(x_2, \text{Caesar}) \lor \text{hate}(x_2, \text{Caesar})$
6. $\text{loyalto}(x_3, f_1(x_3))$
7. $\neg \text{person}(x_4) \lor \neg \text{ruler}(y_1) \lor \neg \text{tryassassinate}(x_4, y_1) \lor \text{loyalto}(x_4, y_1)$
8. $\text{tryassassinate}(\text{Marcus, Caesar})$

9. $\text{Man}(x_5) \lor \text{person}(x_5)$
   A forward-chaining algorithm for propositional definite clauses was already given. The idea is simple: 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**
   First-order definite clauses closely resemble propositional definite clauses they are disjunctions of literals of which exactly one is positive. 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.

This knowledge base contains no function symbols and is therefore an instance of the class DATALOG of Data log knowledge bases—that is, sets of first-order definite clauses with no function symbols

- American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal(x)
- Nono... has some missles
\( \exists x \text{ Owns}(\text{Nono}, x) \land \text{Missile}(x) : \)

\( \text{Owns}(\text{Nono}, M_1) \land \text{Missile}(M_1) \)

- All of its missiles were sold to it by Colonel West

\( \text{Missile}(x) \land \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono}) \)

- We also need to know that missiles are weapons

\( \text{Missile}(x) \Rightarrow \text{Weapon}(x) \)

- and we must know that an enemy of America counts as “hostile”

\( \text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x) \)

- West, who is American”

\( \text{American}(\text{West}) \)

Nono, is a nation

\( \text{Nation}() \text{(nono)} \)

- The country Nono, an enemy of America

\( \text{Enemy}(\text{Nono}, \text{America}) \)

America is a nation

\( \text{Nation()} \text{(America)} \)
function FOL-FC-Ask(KB, α) returns a substitution or false

repeat until new is empty

new ← {}

for each sentence r in KB do

(p₁ ∧ ... ∧ pₙ ⇒ q) ← STANDARDIZE-APART(r)

for each θ such that (p₁ ∧ ... ∧ pₙ)θ = (p’₁ ∧ ... ∧ p’ₙ)θ

for some p’₁, ..., p’ₙ in KB

q’ ← SUBST(θ, q)

if q’ is not a renaming of a sentence already in KB or new then do

add q’ to new

φ ← UNIFY(q’, α)

if φ is not fail then return φ

add new to KB

return false

Analysing the algorithm

Sound

• Does it only derive sentences that are entailed?
• Yes, because only Modus Ponens is used and it is sound

Complete

• Does it answer every query whose answers are entailed by the KB?
• Yes if the clauses are definite clauses

Assume KB only has sentences with no function symbols

• What’s the most number of iterations through algorithm?
• Depends on the number of facts that can be added

  Let k be the arity, the max number of arguments of any predicate and

  Let p be the number of predicates

  Let n be the number of constant symbols

• At most pn^k distinct ground facts
• Fixed point is reached after this many iterations
• A proof by contradiction shows that the final KB is complete

Three sources of complexity

• inner loop requires finding all unifiers such that premise of rule unifies with facts of database
  – this “pattern matching” is expensive
• must check every rule on every iteration to check if its premises are satisfied
• many facts are generated that are irrelevant to goal
Step 1:

Step 2:

Step 3:
Efficiency of forward chaining

- Incremental forward chaining: no need to match a rule on iteration $k$ if a premise wasn't added on iteration $k-1$

  match each rule whose premise contains a newly added positive literal

- Matching itself can be expensive:

  - Database indexing allows $O(1)$ retrieval of known facts

  e.g., query $\text{Missile}(x)$ retrieves $\text{Missile } (M_1)$

Forward chaining is widely used in deductive databases

Backward chaining

In backward chaining, we start from a conclusion, which is the hypothesis we wish to prove, and we aim to show how that conclusion can be reached from the rules and facts in the database. To determine if a decision should be made, work backwards looking for justifications for the decision.

The conclusion we are aiming to prove is called a goal, and the reasoning in this way is known as goal-driven.

Eventually, a decision must be satisfied by facts
function FOL-BC-Ask(KB, goals, θ) returns a set of substitutions
inputs: KB, a knowledge base
goals, a list of conjuncts forming a query
θ, the current substitution, initially the empty substitution {}
local variables: ans, a set of substitutions, initially empty

if goals is empty then return {θ}
q' ← SUBST(θ, FIRST(goals))
for each r in KB where STANDARDIZE-APART(r) = \((p_1 \land \ldots \land p_n \Rightarrow q)\)
and θ' ← UNIFY(q, q') succeeds
ans ← FOL-BC-Ask(KB, [p_1, \ldots, p_n] REST(goals), COMPOSE(θ, θ')) ∪ ans
return ans

The algorithm uses composition of substitution. COMPOSE(θ_1, θ_2) is the substitution whose effect is identical to the effect of applying each substitution is
SUBST(COMPOSE(θ_1, θ_2), p) = SUBST(θ_2, SUBST(θ_1, p))
In the algorithm, the current variable binding, which are stored in θ, are composed with the binding resulting from unifying the goal with the clause head, given a new set of current bindings for the recursive call.

Step 1

Step 2

Step 3
Step 4

Step 5

Step 6
properties of backward chaining
Depth-first recursive proof search: space is linear in size of proof
Incomplete due to infinite loops ⇒ fix by checking current goal against every goal on stack
Inefficient due to repeated subgoals (both success and failure)
⇒ fix using caching of previous results (extra space)
Widely used for logic programming: prolog

Two marks
1 What is Ontological Engineering
   Ontology refers to organizing everything in the world into hierarchy of categories.
   Representing the abstract concepts such as Actions, Time, Physical Objects, and Beliefs is called Ontological Engineering
Define diagnostic rules with example?
Diagnostic rules are used in first order logic for inferencing. The diagnostic rules generate hidden causes from observed effect. They help to deduce hidden facts in the world. For example considering the wumpum world.
The diagnostic rule finding ‘pit’ is,
“If square is breezy some adjacent square must contain pit”, which is written as,

$$\forall x \text{Breezy}(s) \Rightarrow \exists r \text{Adjacent}(r,s) \land \text{pit}(r)$$

<table>
<thead>
<tr>
<th>S. No</th>
<th>Propositional logic</th>
<th>Predicate logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>It is less expressive power</td>
<td>It is more expressive power</td>
</tr>
<tr>
<td>2</td>
<td>It cannot represent relationship among objects</td>
<td>It represent relationship among objects</td>
</tr>
<tr>
<td>3</td>
<td>It does not use quantifiers</td>
<td>It use quantifiers</td>
</tr>
<tr>
<td>4</td>
<td>It does not consider generalization of objects</td>
<td>It consider generalization of objects</td>
</tr>
</tbody>
</table>