AI-Knowledge representation

1.1 Knowledge-based agents

The central component of a knowledge-based agent is its knowledge base KB. A KB is a set of sentences. Each sentence is expressed in a language called knowledge representation language and represents some assertions about the world. When the sentence is taken as being given without being derived is called **axiom.**

To add sentence to the KB or query what is already known there are two operation called *TELL* and *ASK*. Both these operations may involve **inference,** that means deriving new sentences from old ones.

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

The figure above shows the outline of a **knowledge-based agent program**. Each time it is called three things are performed:

- First, it tells the KB what it perceives
- Second, it asks the KB which action to perform
- Last, the program tells the KB which action has been chosen and returns it in order to execute it

Knowledge-based agents are amenable to a description at the knowledge level, where we need to specify only what an agent knows and what its goals are, in order to determine its behavior.

A knowledge-based agent can be built simply by *TELLING* it what it needs to know. Starting with an empty KB, the designer can *TELL* sentences one by one until the agent knows how to operate in its environment. This is called **declarative** approach.

1.2 Logic

Sentences aforementioned are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed.

A logic must also define the **semantics,** or meaning, of sentences. The semantics defines the truth of each sentence with respect to each possible world.

When we need to be precise, we use the term **model** in place of “possible world”.

If a sentence *alpha* is true in model *m*, we say that *m* satisfies *alpha* or sometimes *m* is a model of *alpha*.

Now we can talk about logical reasoning. This involves the relation of logical **entailment** between sentences. The idea that a sentence *follows logically* from another sentence.

$$\alpha \models \beta$$

The formal definition of entailment is this: *alpha entails beta*, if and only if, in every model in which *alpha* is true *beta* is also true. An example from arithmetic: if *alpha* is x=0 this entails *beta* if it is x*y=0.

A possible **logical inference** algorithm is **model checking**, because it enumerates all possible models to check that *alpha* is true in all models in which KB is true, that is M(KB) is contained/equals in M(*alpha*).

If an inference algorithm *i* can derive *alpha* from KB, we write

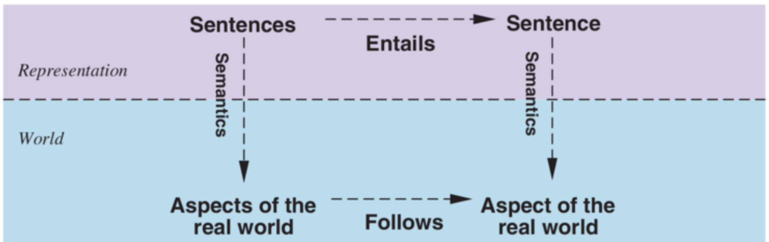
$$KB \vdash \alpha$$

which is pronounced "*alpha* is derived from KB by *I*" or "*i* derives *alpha* from KB"

An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving.** An unsound inference procedure essentially makes is not reliable. Model checking, when applicable, is sound.

The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed.

For what had been described above we have that: if KB is true in real world, the any sentence *alpha* derived from KB by a sound inference procedure is also true in the real world.



Final issue to consider is **grounding**, the connection between logical reasoning processes and the real environment in which the agent exists. We can briefly say that this connection is made through the agent's sensors.

1.3 Propositional Logic

1.3.1 The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false. There are two proposition symbols with fixed meanings:

- *True* which is always true
- *False* which is always false

Complex sentences are constructed from simpler sentences, using parentheses and operators called **logical connectives.** A **literal** is either an atomic sentence (**positive literal**) or a negated atomic sentence (**negative literal**)

There are five main connectives used:

- $\neg$ **not**
- $\wedge$ **and**; a sentence whose main connective is a $\wedge$ is called a conjunction and its parts are **conjuncts**
- $\vee$ **or**; a sentence whose main connective is $\vee$ is called a disjunction and its parts are **disjuncts**
- $\rightarrow$ **implies**; a sentence whose main connective is $\rightarrow$ is called an implication. The sentence which is in the left side of the symbol is called **premise** or **antecedent** and its righter term is called **conclusion** or **consequence**. Implication are also known as **rules** or **if-then** statements
- $\boxed{\leftrightarrow}$ **if and only if**. A sentence containing this as its main symbol is called **biconditional**

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\, \mathit{Sentence} \,) \\
 & | & \neg\, \mathit{Sentence} \\
 & | & \mathit{Sentence} \wedge \mathit{Sentence} \\
 & | & \mathit{Sentence} \vee \mathit{Sentence} \\
 & | & \mathit{Sentence} \;\Rightarrow\; \mathit{Sentence} \\
 & | & \mathit{Sentence} \;\Leftrightarrow\; \mathit{Sentence} \\
 & & \\
\textsc{Operator Precedence} & : & \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

1.3.2 The **semantics** defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply sets the **truth value** for every propositional symbol

$$m = \{P\,(1,2) = \mathit{false},\ P\,(2,2) = \mathit{false},\ P\,(3,1) = \mathit{true}\}$$

The semantics for propositional logic must specify how to compute the truth value of any sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth value of atomic sentences and how to compute the truth of sentences formed with each of the connectives. Atomic sentences are easy:

- *True* is always true and *False* is always false

- The truth value of every other proposition symbol must be specified directly in the model

For complex sentences we have five rules, holding for any subsentences P and Q, in any model *m*.

- ¬P is true iff P is false in *m*
- P∧Q is true iff both P and Q are true in *m*
- P∨Q is true iff either P or Q is true in *m*
- P→Q is true unless P is true and Q is false
- P↔Q is true iff P and Q are both true or both false in *m*

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

1.4 Propositional theorem proving

Theorem proving is an approach to entailment based on the principle of applying rules of inference directly to the sentences in our KB to construct a proof of the desired sentence without consulting models. That's more efficient than model checking if the number of models is large.

Some additional concepts are needed.

Logical equivalence: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as

$$\alpha \equiv \beta$$

Another definition is:

$$\alpha \equiv \beta \text{ if and only if } \alpha \models \beta \text{ and } \beta \models \alpha$$

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee \\
\neg(\neg \alpha) &\equiv \alpha \quad \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg \beta \Rightarrow \neg \alpha) \quad \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg \alpha \vee \beta) \quad \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg \alpha \vee \neg \beta) \quad \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg \alpha \wedge \neg \beta) \quad \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Validity: a sentence is valid if it is true in *all* models. For example, the sentence *P or not P* is valid. Valid sentences are also known as **tautologies**, they are necessarily true. Because the sentence *True* is true in all models, every valid sentence is logically equivalent to *True*.

From the definition of entailment, we can derive the **deduction theorem**:

For any sentences α and β, ($\alpha \models \beta$) if and only if the sentence ($\alpha \rightarrow \beta$) is valid.

Hence, we can decide if $\alpha \models \beta$ by checking that $\alpha \rightarrow \beta$ is true in every model

modusiability: a sentence is satisfiable is it is true in, or satisfied by, *some* model. This property can be checked by simply enumerating all the models until one is found that satisfies the sentence.

Validity and satisfiability are of course related:

α is valid iff $\neg\alpha$ is unsatisfiable; α is satisfiable iff $\neg\alpha$ is not valid

We also have that:

$\alpha \models \beta$ if and only if the sentence $\alpha \wedge \neg\beta$ is unsatisfiable

Proving β from α by checking the un-satisfiability of $\alpha \wedge \neg\beta$ is exactly the standard mathematical proof technique of **reductio ad absurdum.** Also called proof by **refutation** or by **contradiction**.

1.4.1 Inference and proofs

The best-known rule is called **Modus Ponens** and is written:

$$\frac{\alpha \rightarrow \beta,\ \alpha}{\beta}$$

The notation means that, whenever any sentence of the form $\alpha \rightarrow \beta$ and α are given, then the sentence β can be inferred.

Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}$$

One final property of logical systems is **monotonicity**, which says that the sentences can only increase as information is added to the KB. For any sentences α and β:

if $KB \models \alpha$ then $KB \wedge \beta \models \alpha$

1.4.2 Proof by

We want to introduce a single inference rule, **resolution**, that yields a complete inference algorithm when coupled with any complete search algorithm.

???

The **unit resolution** inference rule takes a **clause** and a literal and produces a new clause. Note that a single literal can be viewed as a disjunction of one literal, also known as **unit clause**.

For example known that ($P_{1,1} \vee P_{2,2} \vee P_{3,1}$) and ($\neg P_{2,2}$), this last literal *resolves with* the other statement to give the **resolvent** $P_{1,1} \vee P_{3,1}$

The unit resolution rule can be generalized to the full **resolution rule** which says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses *except* the two complementary literals. For example:

$$\frac{P_{1,1} \vee P_{3,1}, \qquad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}$$

Note that you can resolve only one pair of complementary literals at a time. For example:

$$\frac{P \vee \neg Q \vee R, \qquad \neg P \vee Q}{\neg Q \vee Q \vee R},$$

One last technical aspect is called **factoring** and it resolves the problem that the clause resulting from a resolution should not contain only one copy of each literal. For example if we resolve ($A \vee B$) and ($A \vee \neg B$), we obtain ($A \vee A$), which is reduced to just A by factoring.

Since this rule applies only to clauses, i.e. disjunctions of literals, so it would seem to be relevant only to knowledge bases and queries consisting of clauses. But it is important to know that every sentence of propositional logic is logically equivalent to a conjunction of clauses. A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF**

A **definite clause** is disjunction of literals of which *exactly one* is **positive**.

Slightly more general is the **horn clause**, a disjunction of literals of which *at most one* is **positive.** So, all definite clauses are also horn clauses, as are clauses with no positive literals; these are called **goal clauses**.

Horn clauses are closed under resolution: if you resolve two horn clauses you obtain a horn clause again.

It seems to be useful to show a procedure for converting sentences to CNF. It consists in four steps:

1. elimination of the biconditional so the implication in the two directions.
2. eliminate implication by turning it into disjunction.
3. push negation inside, in other words we want to make sure that the negation appears only in front of atomic symbols and we can do this with De Morgan.
4. a final step of distributing conjunction over disjunction.

Alternative way for steps 2-4

Conjunctive formulae α and disjunctive formulae β :

α	$\alpha_1 \quad \alpha_2$	β	$\beta_1 \quad \beta_2$
$A \wedge B$	$A \quad B$	$A \vee B$	$A \quad B$
$\neg(A \vee B)$	$\neg A \quad \neg B$	$\neg(A \wedge B)$	$\neg A \quad \neg B$
$\neg(A \Rightarrow B)$	$A \quad \neg B$	$A \Rightarrow B$	$\neg A \quad B$

For the transformation in clausal form let F be a propositional formula:

- Step 1
  - let {F} (where F is a set of formulae) be the initial set.
- Step n+1
  - Let the result of step n be {D1, . . ., Dn}, where Di is a disjunction $\{A^i 1, \ldots, A^i k\}$; if $A^i j$ is not a literal (and so it is a formula!), we do not have yet a CNF and we make one of the transformations shown next.

Trasformation in clausal form: algorithm (cont.)

Choose a D_i which contains a non literal X:

a. if X is $\neg\top$ replace it with $\bot$;
b. if X is $\neg\bot$ replace it with $\top$;
c. if X is $\neg\neg A$ replace it with A;
d. if X is a β formula replace it with β_1, β_2;
e. if X is a α formula replace D_i with two clauses:
D_i^1 that is D_i with α replaced by α_1
D_i^2 that is D_i with α replaced by α_2.

◆ Let {L} ∪ D1 and {¬L} ∪ D2 be two clauses. D1 ∪ D2 is obtained from {L} ∪ D1 and {¬L} ∪ D2 by a resolution step written: (similar to modus ponens)

$$\frac{\{L\} \cup D_1 \quad \{\neg L\} \cup D_2}{D_1 \cup D_2}$$

◆ A resolution tree is a binary tree whose nodes are labelled by clauses. Let C1 and C2 be 2 brother nodes, whose father is C, then C1 = D1 ∪ {L}, C2 = D2 ∪ {¬L} and C = D1 ∪ D2. The label associated with the father is the result of a resolution step applied to the sons.

◆ Let Γ be a finite set of clauses, and C a clause. C can be derived by resolution from Γ iff there exists a resolution tree whose root is labelled by C and all the leaves are clauses in Γ. Γ |- R C denotes that Γ derives C by resolution.

Resolution and satisfiability: (for more explanation look the slide, you son of a bitch)

1. let Γ be a set of clauses (conjunction between each clause), {L}∪D1 ∈ Γ and {¬L}∪D2 ∈ Γ. If Γ is satisfiable then Γ ∪ {D1 ∪ D2} is satisfiable;

2. let Γ be a set of clauses, if Γ `R {} then Γ is unsatisfiable.

The empty clause {} in unsatisfiable.

Resolution proofs work by refutation: Γ `R F is proven by checking whether Γ ∪ {¬F} `R {}, where Γ is a set of clauses and {¬F} is the negation of F in clausal form.

Resolution is satisfiability-complete. Let Γ be a set of clauses. Γ is unsatisfiable iff Γ `R {}. Let RC(Γ) denote the resolution closure (i.e. the finite set of clauses that can be derived from Γ using the resolution rule): Γ is unsatisfiable iff {} ∈ RC(Γ)

Resolution sound and complete for propositional logic, but not validity-complete.

1.4.3 A resolution algorithm

Inference procedures based on resolution work by using the principle of proof by contradiction (reductio ad absurdum) introduced in page 5. That is, to show that KB$\models \alpha$, we show that KB$\wedge\neg\alpha$ is **unsatisfiable**. We do this by proving a contradiction.

A resolution algorithm is shown below. First, KB$\wedge\neg\alpha$ is converted into a CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of two things happens:

- There are no new clauses that can be added, in which case KB does not entail α
- Two clauses resolve to yield the *empty* clause, in which case KB entails α

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  while true do
     for each pair of clauses Ci, Cj in clauses do
        resolvents ← PL-RESOLVE(Ci, Cj)
        if resolvents contains the empty clause then return true
        new ← new ∪ resolvents
     if new ⊆ clauses then return false
     clauses ← clauses ∪ new
```

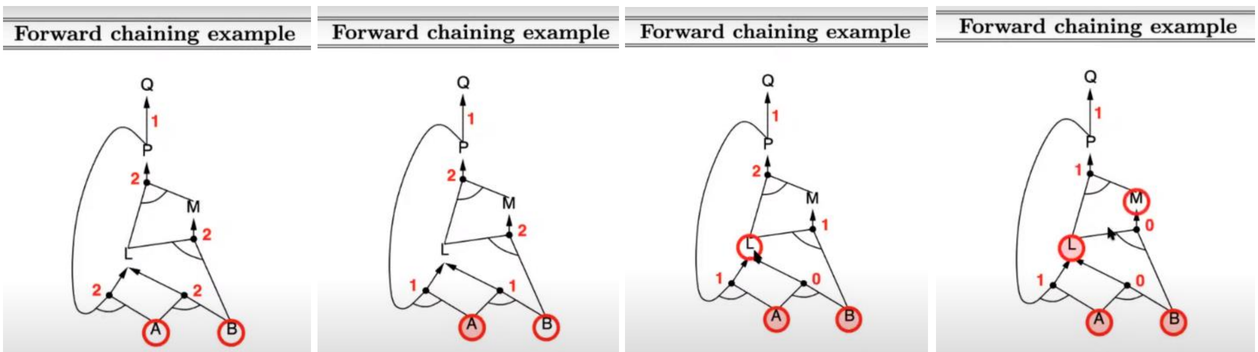
The empty clause, a disjunction of no disjuncts, is equivalent to *False* because a disjunction is true only if at least one of its disjuncts is true. Moreover, the empty clause arises only from resolving two contradictory unit clauses such as P and ¬P.

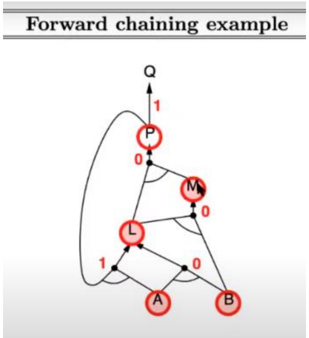
SPIEGAZIONE: Quando voglio verificare che una certa frase α è *entailed* da una certa KB (vera in tutti i modelli in cui è vera KB) e so che al momento la KB è *satisfiable* (esiste almeno un modello M tale che KB è vera in M), provo ad aggiungere $\neg\alpha$ alla KB e vedere se rimane soddisfabile. Se si, allora α non è *entailed* dalla KB (in sostanza è falsa per quella KB); se no, allora α è *entailed* dalla KB. Mi accorgo che (KB $\wedge \neg\alpha$) non è più soddisfabile quando, mentre espando il *resolution tree*, eseguendo la *resolution* tra le *clauses* ottengo la *empty clause* { }. Questo perché essendo nella CNF tutte le *clauses* messe in *and*, ed essendo la *empty clause* **non soddisfabile per definizione** sicuramente non posso trovare un modello che soddisfa KB $\wedge \neg\alpha$.

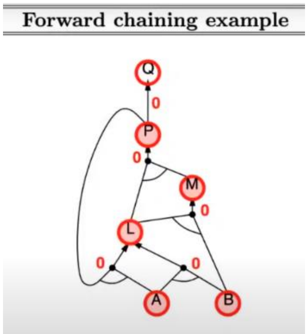
1.4.4 Forward and Backward chaining

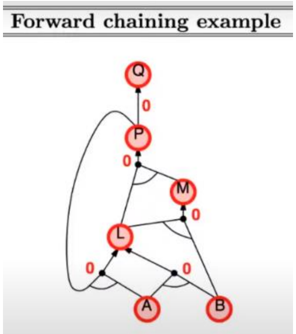
Forward chaining determines if a single proposition symbol q—the query—is entailed by a KB of definite clauses. It begins from known facts (positive literals) in the KB. If all the premises of an implication are known, then its conclusion is added to the set of known facts.

This process continues unctil the query q is added or until no further inferences can be made.







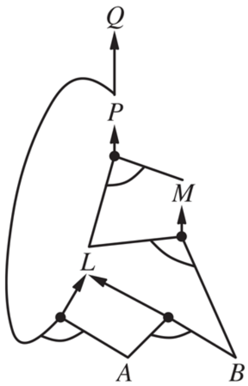


The best way to understand the algorithm is through an example and a picture. Figure shows a simple knowledge base of Horn clauses with and as known facts. Figure shows the same knowledge base drawn as an AND–OR graph. In AND–OR graphs, multiple edges joined by an arc indicate a conjunction—every edge must be proved—while multiple edges without an arc indicate a disjunction—any edge can be proved.

The known leaves (here, A and B) are set, and inference propagates up the graph as far as possible. Wherever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding.

$$
\begin{aligned}
&P \Rightarrow Q\\
&L \land M \Rightarrow P\\
&B \land L \Rightarrow M\\
&A \land P \Rightarrow L\\
&A \land B \Rightarrow L\\
&A\\
&B
\end{aligned}
$$

(a)



(b)

Forward chaining is sound (by modus ponens approach), and also complete.

Forward chaining is an example of the general concept of data-driven reasoning—that is, reasoning in which the focus of attention starts with the known data.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is initially the number of symbols in clause c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  queue ← a queue of symbols, initially symbols known to be true in KB

  while queue is not empty do
     p ← POP(queue)
     if p = q then return true
     if inferred[p] = false then
         inferred[p] ← true
         for each clause c in KB where p is in c.PREMISE do
             decrement count[c]
             if count[c] = 0 then add c.CONCLUSION to queue
  return false
```

The backward-chaining algorithm, as its name suggests, works backward from the query q. If the query is known to be true, then no work is needed. Otherwise, the algorithm finds those implications in the knowledge base whose conclusion is q. If all the premises of one of those implications can be proved true (by backward chaining), then is true. When applied to the query, it works back down the graph until it reaches a set of known facts, A and B, that forms the basis for a proof.

Avoid loops: check if new subgoal is already on the goal stack. Avoid repeated work: check if new subgoal:

1) has already been proved true, or

2) has already failed

Backward chaining is a form of goal-directed reasoning. As with forward chaining, an efficient implementation runs in linear time (often it takes less than linear time).

----------------------------------MISSING SOME ARGUMENTS???-------------------------------------

2. First Order Logic

Whereas propositional logic assumes world contains **facts**, first-order logic assumes the world contains

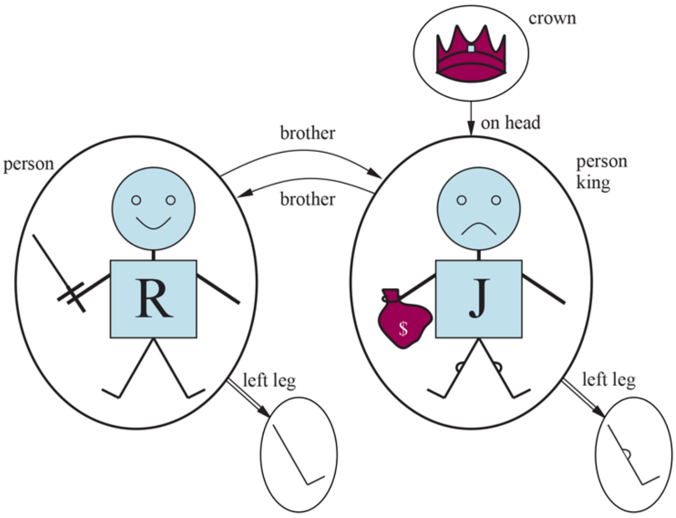
- Objects: people, houses, numbers
- Relations: (unary) red, round…; (n-ary) brother of, bigger than…
- Functions: father of, best friend…

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

We've said that models for propositional logic are the formal structures that constitute the possible worlds under consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined.

Instead, models for FOL are much more interesting (che culo). First, the 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 *non-empty*.

We will use the following example containing five objects, two binary relations (brother and on-head), three unary relations (person, king and crown) and one unary function (left-leg).



Formally speaking, a relation is just the set of **tuples** of objects that are related. Thus, the *brotherhood* relation in this model is the set:

{<R, J>, <J, R>}

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, a mapping from a one-element tuple to an object.

<R> -> R's left leg

Strictly speaking, models in FOL require **total functions**, that is, there must be a value for every input tuple. Thus, the crown must have a left leg and so must each of the left legs. There is a technical solution to this awkward problem involving an additional "**invisible**" object that is the left leg of everything that has no left leg, including itself.

2.1 Syntax of FOL

We turn now to the syntax of FOL:

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{Predicate} \mid \textit{Predicate}(\textit{Term},\ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} & \rightarrow & (\ \textit{Sentence}\ ) \\
 & \mid & \neg\, \textit{Sentence} \\
 & \mid & \textit{Sentence} \land \textit{Sentence} \\
 & \mid & \textit{Sentence} \lor \textit{Sentence} \\
 & \mid & \textit{Sentence} \ \Rightarrow\ \textit{Sentence} \\
 & \mid & \textit{Sentence} \ \Leftrightarrow\ \textit{Sentence} \\
 & \mid & \textit{Quantifier Variable},\ldots\ \textit{Sentence} \\
 & & \\
\textit{Term} & \rightarrow & \textit{Function}(\textit{Term},\ldots) \\
 & \mid & \textit{Constant} \\
 & \mid & \textit{Variable} \\
 & & \\
\textit{Quantifier} & \rightarrow & \forall \mid \exists \\
\textit{Constant} & \rightarrow & A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\textit{Predicate} & \rightarrow & \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} & \rightarrow & \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\textrm{OPERATOR PRECEDENCE} & : & \neg, =, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

The basics elements on FOL are the symbols that stand for objects, relations, and functions. The symbols, therefor, come in three kinds:

- **Constant symbols**, which stand for objects
- **Predicate symbols**, which stand for relations
- **Function symbols**, which stand for functions

According to the convention these symbols will begin with uppercase letters.

Each predicate and function symbol comes with an **arity** that fixes the number of arguments. Every model must provide the information required to determine if any given sentence is true or false. Thus, in addition to its objects, relations and functions each model includes an **interpretation** that specifies exactly which objects, relations and functions are referred to by the constant, predicate and function symbols. One possible interpretation for our example, which a logician would call the **intended interpretation,** is as follows:

- *R* refers to Richard the Lionheart and *J* refers to the evil king John
- *Brother* refers to the brotherhood relation…
- *Left-Leg* refers to the "left leg" function previously defined

In summary, a model in first-order logic consists of a set of objects and an interpretation that
maps constant symbols to objects, function symbols to functions on those objects, and predicate symbols to relations. Just as with propositional logic, entailment, validity, and so on are defined in terms of *all possible models*.

2.1.1 Terms and Atoms

A **term** is a logical expression that refers to an object. Every constant and variable symbol are terms. If *t1…tn* are terms and *f* is an n-ary function symbol, *f (t1…tn)* is a term (functional term).

An **atomic sentence** or **atom** is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as: *Brother (R, J)*

Atomic sentences can have complex terms as argument

Married (Father(R), Mother(J))

We usually follow the argument-ordering convention that P (x, y) is read as "x is a P of y"

A term denotes an object of our world, the atom denotes a property of objects (expresses a truth value, so could be true or false)

2.1.2 Complex sentences- logical connectives and quantifiers

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

Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating objects by name. **Quantifiers** let us do this. In FOL we have two standard quantifiers, called **universal (∀)** and **existential (∃)**.

FOL includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the **equality symbol** to signify that two terms refer to the same object.

2.1.3 Using first order logic (FOL)

Sentences are added to a KB using **tell,** as in propositional logic. Such sentences are called **assertions**. For example, we can assert that John (J) is a king

TELL (KB, King(J))

We can ask questions of the KB using **ask**. For example:

ASK (KB, Person(J))

And this returns true. We can also ask quantified queries.

But if we don't want a Boolean answer, we have to use a different function called **askvars (ask variables)**:

ASKVARS(KB, Person(x))

Which yields a stream of answers. In this case there will be two answers: *{x/J}* and *{x/R}*

Such an answer is called a **substitution** or **binding list**.

Axioms are commonly associated with purely mathematical domains, but they are needed in all domains. They provide the basic factual information from which useful conclusions can be derived.

2.2 Inference in FOL by propositionalisation

One way to do first-order inference is to convert the first-order KB to propositional logic and use propositional inference. A first step is eliminating universal quantifiers, in case of universal one, by enumerating all the statements wrapped with the quantifier.

In general, the rule of **universal instantiation (UI)** says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for a universally quantified variable.

To write out the inference rule formally, we use the notion of **substitutions.** Let SUBST (θ, α) denote the result of applying the substitution θ to the sentence α. Then the rule is written:

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

Similarly, the rule of **existential instantiation** replaces an existentially quantified variable with a single new constant symbol.

$$\frac{\exists v\ \alpha}{\text{SUBST}(\{V/G\}, \alpha)}$$

A **skolem constant** is the name of an object that satisfies a certain statement, but this name does not already belong to another object contained in the KB.

After having removed all the quantifiers, and replaced all the atomic sentences with proposition symbols, we can finally apply any of the complete propositional algorithms (missing in this file) to obtain conclusions.

This technique is called **propositionalisation**. A problem with this could be that if the KB includes a function symbol, the set of possible ground term substitutions is infinite. For example, if the KB mentions the *Father* symbol, then infinitely many nested terms such as *Father(Father(Father(John)))* can be constructed.

But there exists a theorem stating that if a sentence is entailed by the original, first-order KB, then there is a proof involving just a *finite* subset of the propositionalised KB. (???)

2.3 Unification and First-Order Inference

Consider the universal quantified statement ∀x *King(x)* ∧ *Greedy(x)* → *Evil(x)* in the KB and the other statements in the KB:

- *King (John)*
- *Greedy (John)*
- *Brother (Richard, John)*

As noticed, the propositionalisation approach generates many unnecessary instantiations of universally quantified sentences.

King (John) ∧ *Greedy (John)* → *Evil (John)*

King (Richard) ∧ *Greedy (Richard)* → *Evil (Richard)*

We'd rather have an approach that uses just the one rule, reasoning that *{x/John}* solves the query *Evil(x)* as follows: given the rule that greedy kings are evil, find some x such that x is a king and x is greedy, and then infer that this x is evil. More generally, if there is some substitutions θ that makes each of the conjuncts of the premise of the implication identical to sentence already known in the KB, then we can assert the conclusion of the implication, after applying θ. In this case, the substitution θ = *{x/John}* achieves that aim.

This inference process is called **Generalized Modus Ponens**.

$$\frac{p_1{}',\ p_2{}',\ \ldots,\ p_n{}',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\textsc{Subst}(\theta,q)}.$$

In 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 $Evil(x)$
$\textsc{Subst}(\theta,q)$ is $Evil(John)$.	

This is a sound inference method.

2.3.1 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 algorithm. The UNIFY algorithm takes two sentences and returns a **unifier** for them (a substitution) if one exists:

UNIFY (p, q) = θ where SUBST (θ, p) = SUBST (θ, q)

2.3.2 Resolution in FOL

Resolution is the only logic system which works on every kind of KB, not only on the Horn ones. As already said for applying resolution we need to convert our KB in CNF and, due to the fact that in FOL we have also variables and quantifiers, the procedure is a bit different.

Here, we have that each literal contained in the disjunctions can contain variables which are supposed to be universally quantified. For example:

$$\forall x,y,z \;\; American(x) \land Weapon(y) \land Sells(x,y,z) \land Hostile(z) \Rightarrow Criminal(x)$$

Which becomes in CNF (c'è scritto not American al primo literal, si vede poco ma non mi va di rifare il ***taglio [del samuele]***):

$$\neg American(x) \lor \neg Weapon(y) \lor \neg Sells(x,y,z) \lor \neg Hostile(z) \lor Criminal(x).$$

The key is that every sentence of FOL can be converted into an inferentially equivalent CNF sentence. The procedure, which is similar to the one seen for propositional logic, is different due to the precence of quantifiers. It is illustrated through the example "Everyone who loves animals is loved by someone(animal)", or:

$$\forall x \; [\forall y \; Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y \; Loves(y,x)].$$

The steps are as follows:

- **Eliminate implications**: replace P -> Q with ¬P∧Q. In our example:

$$\forall x \quad \neg[\forall y \quad Animal(y) \Rightarrow Loves(x,y)] \lor [\exists y \quad Loves(y,x)]$$
$$\forall x \quad \neg[\forall y \quad \neg Animal(y) \lor Loves(x,y)] \lor [\exists y \quad Loves(y,x)].$$

- **Move ¬ inwards**: In addition to the usual rule for negated connectives, we need rules for negated quantifiers. Thus, we have:

$$\begin{array}{ccc} \neg\forall x \; p & \text{becomes} & \exists x \; \neg p \\ \neg\exists x \; p & \text{becomes} & \forall x \; \neg p. \end{array}$$

(Non tutti p, quindi esiste un non p. Non esiste un p, quindi tutti non p)

In our example:

$$\forall x \; [\exists y \quad \neg(\neg Animal(y) \lor Loves(x,y))] \lor [\exists y \quad Loves(y,x)].$$
$$\forall x \; [\exists y \quad \neg\neg Animal(y) \land \neg Loves(x,y)] \lor [\exists y \quad Loves(y,x)].$$
$$\forall x \; [\exists y \quad Animal(y) \land \neg Loves(x,y)] \lor [\exists y \quad Loves(y,x)].$$

Note that, even if the universal quantifier in the square parenthesis has been replaced with an existential one, the meaning of the sentence has been preserved through the negation.

- **Standardize variables**: For sentences that use the same variable name twice, change the name of one the variables. This avoids confusion later when we will drop the quantifiers. Thus, we have:

$$\forall x \ [\exists y \ \ Animal(y) \land \neg Loves(x,y)] \lor [\exists z \ \ Loves(z,x)].$$

- **SKOLEMIZE: Skolemization** is the process for removing existential quantifiers by elimination. In the simple case, it is just like the Existential Instantiation rule previously mentioned: translate *∃x P(x)* into *P(A)* where A is a new constant (new means that there must be no other constants with that name in the KB). However, we cannot apply existential instantiation rule to our sentence because it does not match the pattern ∃v α (in fact, there is an *and* after the quantifier and not a simple literal in the first square parenthesis). Applying blindly the rule to the two matching part we get:

$$\forall x \ [Animal(A) \land \neg Loves(x,A)] \lor Loves(B,x) \ ,$$

which has the wrong meaning. So, we want to Skolem entities to depend on x:

$$\forall x \ [Animal(F(x)) \land \neg Loves(x,F(x))] \lor Loves(G(x),x).$$

Here, F and G are **Skolem Functions.** The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original is satisfiable.

- **Drop universal quantifiers**: at this point, all remaining variables must be universally quantified. Therefore, we will not lose any information if we drop the universal quantifiers:

$$[Animal(F(x)) \land \neg Loves(x,F(x))] \lor Loves(G(x),x).$$

- **Distribute ∨ over ∧**:

$$[Animal(F(x)) \lor Loves(G(x),x)] \land [\neg Loves(x,F(x)) \lor Loves(G(x),x)].$$

Skolem function F(x) refers to the animal potentially unloved by x, whereas G(x) refers to someone who might love x.

Resolution for FOL is simply a lifted version of the propositional one. Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain complementary literals.

Example of resolution:

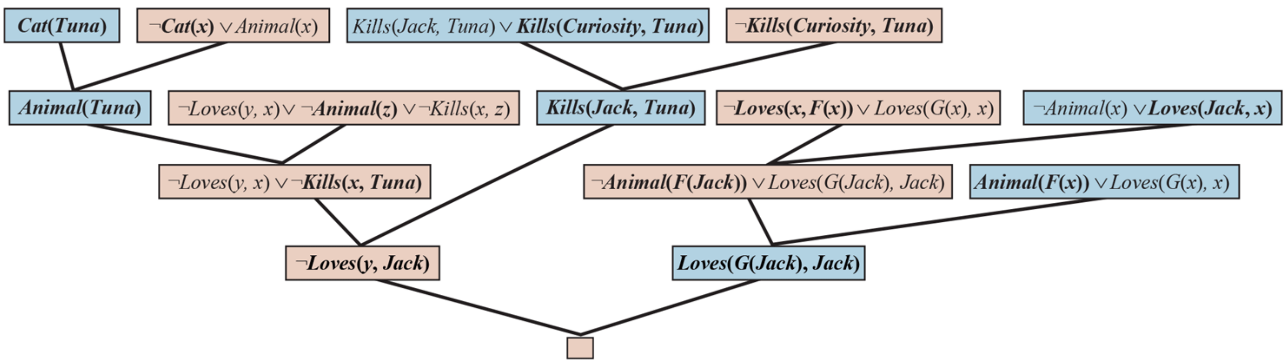
First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

$$
\begin{array}{rl}
\text{A.} & \forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)] \\
\text{B.} & \forall x\ [\exists z\ Animal(z) \land Kills(x,z)] \Rightarrow [\forall y\ \neg Loves(y,x)] \\
\text{C.} & \forall x\ Animal(x) \Rightarrow Loves(Jack,x) \\
\text{D.} & Kills(Jack,Tuna) \lor Kills(Curiosity,Tuna) \\
\text{E.} & Cat(Tuna) \\
\text{F.} & \forall x\ Cat(x) \Rightarrow Animal(x) \\
\neg\text{G.} & \neg Kills(Curiosity,Tuna)
\end{array}
$$

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

$$
\begin{array}{rl}
\text{A1.} & Animal(F(x)) \lor Loves(G(x),x) \\
\text{A2.} & \neg Loves(x,F(x)) \lor Loves(G(x),x) \\
\text{B.} & \neg Loves(y,x) \lor \neg Animal(z) \lor \neg Kills(x,z) \\
\text{C.} & \neg Animal(x) \lor Loves(Jack,x) \\
\text{D.} & Kills(Jack,Tuna) \lor Kills(Curiosity,Tuna) \\
\text{E.} & Cat(Tuna) \\
\text{F.} & \neg Cat(x) \lor Animal(x) \\
\neg\text{G.} & \neg Kills(Curiosity,Tuna)
\end{array}
$$

And the resolution tree:



3 Prolog

Declarative Programming

- Program = problem description -> KB

- Execution = proving that something follows from the KB; check the truth of an assertion (goal) through inference(?)

Basic intuition:

1. Definition of the problem through the assertion of **facts** and **rules**
2. Querying the system which **infers** the answer to the query given known facts and rules

Example: Aristotelic Syllogism in PROLOG

mortal (X): - man (X) which means "man implies mortal"

man (Socrates)

The inference is started by

?mortal (Socrates)

And prolog will answer yes.

A PROLOG program is composed by a set of clauses, i.e. *conditional* (**rule**)and *unconditional* (**fact**) assertions

A fact is father (daniele, jacopo).

But also loves (enzo, X)

In PROLOG the names of predicates and constants start with lower letter

Names of variables with capital one.

Rules:

A:- B,C, ... D.

Commas stands for *and*, the set is a conjunction.

A is true if B, C, … D are true.

- A is the **conclusion (head)**
- B, C, …, D are the **premises (body)**
- A, B, C, D are **atoms**

Atoms are predicate symbols applied to arguments. Arguments (**terms**) can be either variables or constants.

General form of a predicate is P (t_1, ... t_n) where t_1, ... t_n are (for the time being) only terms.

PROLOG

PROLOG is the major logic-based programming language (subset of First Order Logic).

Applications of logic programming:

- deductive databases
- expert systems
- knowledge representation for robots

Logic program:

1. Definition of the problem through the assertion of facts and rules.
2. Querying the system which infers the answer to the query given known facts and rules (theorem provers).

For example:

All men are mortal, Socrates is a man → we can infer: Socrates is mortal.

mortal(X) :- man(X).

$:-$ has two meaning depending on the verse in which you read it:

- Left to Right = (x mortal is true) **if** (x man is true)
- Right to Left = **implies** (if x is a man, then x is mortal)

man(socrates).

A PROLOG program is composed by a set of clauses, i.e. conditional and unconditional assertions.

Unconditional assertion (**fact**) → $father(daniele, jacopo).$

$father = predicate\ symbol.\ \ daniele\ and\ jacopo\ are\ argumets.$

$father(daniele, jacopo)$ First arg is usually the subject (Daniele is the father of Jacopo).

In PROLOG:

- the names of predicates and constants/individuals start with a lower-case letter (e.g. father, jacopo),

- while the variable identifiers start with a capital letter (e.g. X).

Conditional assertion (**rule**):

A :- B,C,...,D.

A is true if B, C,..., D are true,

- A is the conclusion,
- B, C,... , D are the premises
- A,B,C, D are atoms

If t1 , . . . , tn are terms and **P** is an n-ary predicate P(t1 , . . . , tn) is an atom. We start simple (without function symbols), so terms are either constants or variables.

Logica

Quando hai **l'entails** o **l'implies** il problema nasce (**false**) quando il **primo predicato** è **vero** (**1**) mentre il secondo è falso(**0**), tutti gli altri casi vanno bene (**true**).