Knowledge Representation Summary

Author: **Nicolò Brandizzi**

Contributors:

DIAG
Sapienza
November 2018

Contents

Abstract

This is **free** material! You should not spend money on it.
This notes are about the *Knowledge Representation* part taught by professor Daniele Nardi in the Artificial Intelligence class. Everyone is welcome to contribute to this notes in any relevant form, just ask for a pull request and be patient.
Remember to add your name under the contributors list in the title page when submitting some changes (if you feel like it).

1 Logic based agents

A **knowledge base** [KB] is a set of sentences in a knowledge representation language that express some assertion about the world.
We can either:

- *Tell*: i.e. add new sentences to the KB or
- *Ask*: i.e. query what is known.

We can use both this actions to do **inference** in which we derive new sentences from old ones.

There are two ways of building a KB structure:

- **Declarative**: in which the agent is *told* various aspects of the world [1] until it is capable of working in the environment.
- **Procedural**: which encodes desired behavior directly in the program.

Finally an agent can be viewed at:

- *Knowledge level*, where we specify what the agents knows and what are its goals. Or at
- *Implementation level*, where we need to specify the data structures used in the KB and the way to manipulate them.

1.1 Logic

Sentences must follow two principles in order to be considered *correct*:

- **Syntax** holds when a sentence is *well formed*, e.g. in mathematics "$x+y = 4$" is correct while "$x3y+ =$" is not.
- **Semantic** that defines the *truth* of a sentence in respect to each possible world. For example the sentence $x + y = 4$ is true in a world where $x = 2, y = 2$ but false when $x = 1, y = 4$.

Other than saying *each possible world* when referring to the KB, we can use the word **interpretation**. Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, interpretations are mathematical abstractions, each of which simply fixes the truth

[1] That is the designer add new sentences.

or falsehood of every relevant sentence.

Moreover a **model** m is an interpretation of a sentence α if α is true in m [2]. Given a set of the models of α, $M(\alpha)$, we can derive the concept of *entailment*: A KB *entails* a sentence α [3] if and only if, in every interpretation in which the KB is true [4], α is also true:

$$KB \models \alpha \Leftrightarrow M(KB) \subseteq M(\alpha)$$

Note that $M(KB) \subseteq M(\alpha)$ means that KB is a *stronger assertion* than α since it *rules out* more interpretation or possible worlds [5].

Example Lets make an example to better understand this concepts.
We have the following sentences:

- α = Overwatch is better than Fortnite.
- β = Everything is better than Diablo immortal [6].

Our KB is equal to $KB = \alpha \wedge \beta$, that is the KB knows that *Overwatch is better than Fortnite **and** that Diablo immortal sucks.* We want to know:

$$KB \models \beta$$

i.e. we can derive from KB that Diablo immortal sucks.
We first need to check if $M(KB) \subseteq M(\beta)$, in other words if the KB has fewer true interpretation [7] than β. We know that the entailment holds since:

- The KB has two independent sentences, α, β, correlated by an *and* logic relationship, which result in fewer models.
- We are asking the KB for the truth of a sentence which is directly part of the KB itself; so that the set of models $M(KB)$ is a *subset* of $M(\beta)$.

Model checking To know if $KB \models \beta$ we need to prove that $M(KB) \subseteq M(\beta)$. This can be done by **model checking**, that is enumerating all the possible interpretation where KB is true and check if those are also models of β [8]. For the above example we have:

	KB	β
$\alpha \wedge \beta$	True	True
$\neg\alpha \wedge \beta$	False	True
$\alpha \wedge \neg\beta$	False	False
$\neg\alpha \wedge \neg\beta$	False	False

Table 1: Model checking for $KB \models \beta$

[2] Here the Russell-Norvig has a different concept of *model* which is equal to the above *interpretation (= each possible world = model)* . But Nardi prefer to make the distinction between the two so we will go with the Nardi flow (*each possible world=interpretation ≠ model*).
[3] That is the sentence α follows logically/is derived from KB.
[4] That is in every *model* of the KB.
[5] There are less models in KB than in α.
[6] Don't you have phones?
[7] Models.
[8] This is a direct implementation of the definition of entailment.

Deduction To better understand the difference between entailment and inference we should think of the *set of all consequences* of KB as a haystack and β as a needle. Entailment is like the needle being in the haystack; inference is like finding it.

Another way of computing the knowledge entailed by a KB is by a **deduction procedure**:

$$KB \vdash_i \beta$$

Which denotes that β can be driven from KB by an **inference algorithm** i.

Sound and Completeness Given an inference algorithm i, if i derives **only entailed** sentences from KB then it is considered **sound** [9], otherwise i would make things up as it goes along (discovering of non-existing needles).
On the other hand **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. For many knowledge bases, however, the haystack of consequences is infinite, and completeness becomes an important issue.

Difference between $\models$ and $\vdash$ Having

- $KB \models \alpha$
- $KB \vdash \alpha$

The first symbol $\models$ is called **entailment** and means that α must be true in all of KB's models, that is KB are true when α's interpretations are true [10].
The other symbol $\vdash$ is read **derives** (KB derives α) it can be joint with a sign denoting an inference rule, such as $KB \vdash_{MP} \alpha$ [11] or $KB \vdash_R \alpha$ [12] or both $KB \vdash_{R,MP} \alpha$. You can even ignore the set of inference rules and just write $KB \vdash \alpha$, in this case you are saying that *it must exist a derivation* [13] *so that α is the last element of the chain.* You can verify the entailment with the derivation if and only if the inference rules you are applying are sound and complete.

Difference between *deduction* and *inference* Broadly speaking they are the same thing.
More specifically the *deduction* is always referred to as a syntactic derivation, while *inference* is a more generic term which means that starting from KB we can conclude α. So *deduction* is tied to the $\vdash$ symbol, while *infer* is more generic and can be used in both $\vdash, \models$

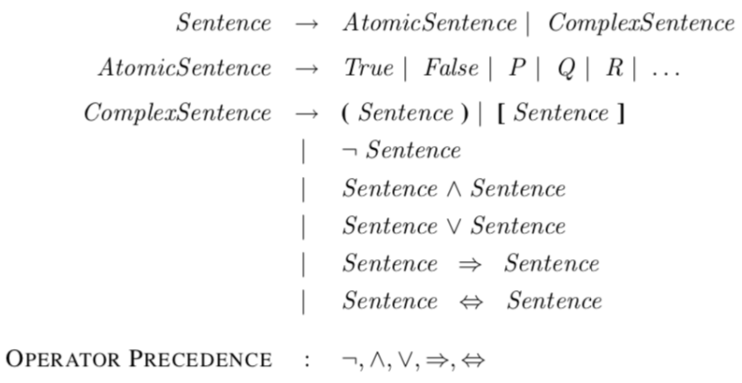
[9] Or truth preserving.
[10] We can have some models of α which are not model in KB.
[11] Modus Ponens Section 2.1.2.
[12] Resolution Section 2.2.
[13] Obtained with some inference rule.

2 Propositional Logic

Propositional logic is the simplest logic! [14]

Syntax An **atomic sentence** is made of *one* **propositional symbol** (for example S), which can be either *True or False.*
True and False are propositional symbols that are always True/False.
Complex sentences are propositional symbols joint together by the following **logical connective** (Figure 1) :

- $\neg$ (not): $\neg S$ is the **negation** of S.
- $\wedge$ (and): $S_1 \wedge S_2$ is a **conjunction**.
- $\vee$ (and): $S_1 \vee S_2$ is a **disjunction**.
- $\Rightarrow$ (implies): $S_1 \Rightarrow S_2$ is an **implication**, where S_1 is the *antecedent/premise* and S_2 is the *consequent/conclusion.* Implication are known as **if-then** statement [15].
- $\Leftrightarrow$ (if and only if): $S_1 \Leftrightarrow S_2$ is a **biconditional**.



$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \ldots \\
\textit{ComplexSentence} & \rightarrow & (\ \textit{Sentence}\ ) \mid [\ \textit{Sentence}\ ] \\
 & \mid & \neg\ \textit{Sentence} \\
 & \mid & \textit{Sentence} \wedge \textit{Sentence} \\
 & \mid & \textit{Sentence} \vee \textit{Sentence} \\
 & \mid & \textit{Sentence} \Rightarrow \textit{Sentence} \\
 & \mid & \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
 & & \\
\text{OPERATOR PRECEDENCE} & : & \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 1: Syntax summary for propositional logic

Semantic The semantic defines the truth of a sentence in respect to a particular interpretation, that assign a truth value to every propositional symbol (Figure 2).

[14]Not to understand...
[15]If premise then conclusion.

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*

Figure 2: Truth Table for propositional logic.

For what regard the *implies* symbol ($P \Rightarrow Q$) we need to specify that:

- It *does not* need to have any causal/relevance link between the antecedent and the consequent. For example *'5 id odd implies Tokyo is the capital of Japan'* is syntactically correct.
- Any implication is true whenever the antecedent is false. This because we are saying *'If P is true, then I am claiming that Q is true. Otherwise I am making no claim'*.
- The only way for $P \Rightarrow Q$ to be false is if P is true and Q is false.

Inference Our goal is to prove that

$$KB \models \alpha$$

With the model checking approach we just need to enumerate all the possible interpretation and check that, when there is a model of KB then there is also a model of α. This is done by assigning either *true* or *false* to every propositional symbol in any interpretation. But given n symbols there are 2^n interpretations, thus the time complexity is $O(2^n)$, while the space complexity is $O(n)$ since we're using a depth-first approach.

2.1 Theorem Proving

We can use a technique known as *theorem proving* that consist in applying rules of inference directly to the sentences in our KB to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking. We first need to introduce some concepts.

Logical equivalence Two sentences α and β are logically equivalent if they are true in the same set of interpretations, i.e. they have the **same set of models**. We write this as $\alpha \equiv \beta$. We can formalize this propriety by writing:

$$\alpha \equiv \beta \Longleftrightarrow \alpha \models \beta \wedge \beta \models \alpha$$

Following there are some standard logic equivalences (Figure 3):

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

Figure 3: Standard logic equivalences.

Validity A sentence is *valid* if it is true in all interpretations [16]. For example: $A \vee \neg A$, $True$, $A \Rightarrow A$...
Validity can be tied to the deduction problem in the following way:
For every sentences α and β, $\alpha \models \beta$ if and only if the sentence $\alpha \Rightarrow \beta$ is valid.

Satisfiability A sentence is **satisfiable** [17] if it is true in *some* interpretation, i.e. it has some models. A sentence can be proved to be satisfiable by enumerating the possible implementations until a model is found [18].
We can say that:

- *α is valid iff $\neg\alpha$ is unsatisfiable*, or
- *α is satisfiable iff $\neg\alpha$ is not valid.*

Hence we can give another interpretation for entailment:

$$\alpha \models \beta \Longleftrightarrow (\alpha \wedge \neg\beta) \quad is \quad unsatisfiable$$

Which is also known as the **reductio ad absurdum** (proof by contradiction).

2.1.1 Deduction in Propositional Logic

We can use *inference rules* to derive a proof [19] of the interpretation truthfulness. The idea behind finding a proof rather than using model checking is that the proof can ignore irrelevant proposition, no matter how many of them there are. Usually this kind of rules are written in the form:

$$\frac{premisies}{conclusions} = \frac{A_1, A_2, ..., A_n}{A}$$

[16] Also known as tautologies.
[17] This problem is called SAT and it has been shown to be NP-complete.
[18] Model checking technique.
[19] A proof is a chain of conclusion which leads to a desired goal.

Suppose we want to derive some formula [20] α from the KB[21] ($KB \vdash \alpha$), there must be a sequence of formulas $\alpha_1, ..., \alpha_n$ such that:

- For every i between $1...n$ either:
  - $\alpha_i \in KB$, that is the formula α_i is in the KB. Or
  - α_i is a *direct derivation* of $\alpha_{i-k},\ k \in [1, i-1]$ [22]
- $\alpha = \alpha_n$, the chain of direct derivations brings to the formula we want to infer.

Hence we can say that $\alpha_1, ..., \alpha_n$ is a proof of α from the KB. Finding a proof for a formula can implemented as a search where:

- *Initial State*: is the KB .
- *Operators*: are *inference rules* (mentioned earlier).
- *Final state*: is the formula to be proven.

Basic proprieties We need to describe the proprieties of a inferring method $\mathcal{R}$ given a set of formulas KB and a formula A. It is important to understand that by writing $\models A$ we are referring to the truthfulness of A in all its interpretation, thus referring to the *validity* of A.

- First we need to prove that an infer rule $\mathcal{R}$ is **sound**, to do so we need to prove that A is **valid** given the fact that it can be derived with $\vdash_{\mathcal{R}}$.
  First thing is to notice that there is no KB before the symbol since we want to prove A being valid regardless of the existence of any KB [23].
  But if we can derive A with a deduction $\vdash$ then A must be valid, so $\mathcal{R}$ is **sound** and we have: $KB \vdash_{\mathcal{R}} A$ implies $KB \models A$.
- On the other hand, if A is valid then it exist a derivation $\vdash_{\mathcal{R}}$ than let me derive it, so that $KB \models A$ implies $KB \vdash_{\mathcal{R}} A$, thus $\mathcal{R}$ is **complete**.

2.1.2 Inference and proof

First thing first the truth tables we cited before are not associated with the inference rules. They are associated with the formulae! So you cannot apply any truth table to Modus Ponens, And Elimination and Resolution since it does not make sense.

Moreover atom and literal are the two basic elements for the construction of the formulae, in propositional logic A is a propositional variable. But since we can have bot A and $\neg A$ we say that A is a literal which can be positive or negative.

[20] Or sentence.
[21] Which is a set of formulas.
[22] α_i is a direct derivation of the previous formulas. In general α_i can depend on any subset of previous α, it depends on the resolution rule used.
[23] That is a formula which is true in all models, for example $A \vee \neg A$.

The syntax (Section 2) allows you to build a formula regardless of the amount of free variable [24], when you have a formula with no free variables then its called a **sentence**. For example if we say:

$$A \wedge B \Rightarrow C$$

then its a formula, but if we associate a meaning to each literals:

$$Student \wedge SteamSales \Rightarrow Happy$$

Then it becomes a sentence.

On the other hand we used atom to indicate a formula for the estimations of predicates that has a predicates and some arguments. While in the propositional logic we have the propositional variable as a base element and an atom is a literal, in First Order Logic we have atoms.

Modus Ponens We can ether use Modus Ponens:

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

Which given $\alpha \Rightarrow \beta$ and α it can infer β. For example

$$\frac{man \Rightarrow mortal, \quad man}{mortal}$$

or

$$\Gamma = \{feline \Rightarrow animal, cat \Rightarrow feline, cat\}$$

Result in

$$\Gamma \vdash_{MP} animal$$

Which mean that *animal* can be derived from Γ using the inference algorithm MP (Modus Ponens).

And Elimination On the other hand we can use and elimination:

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

more in general:

$$\frac{\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n}{\alpha_i}$$

Which works for any chain of conjunction and means that *"if the rule $A = \{\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n\}$ is true, then for any $\alpha_i \in A$, α_i must be true as well"*, since we have an *and* logical chain.

[24]This is use in general cases.

Monotonicity Finally **monotonicity** is the propriety of logical system which says that the set of entailed sentences can only increase as information is added to the knowledge base:

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

Which means that inference rules can be applied whenever suitable premises are found in the KB; conclusion of the rule must follow regardless of what else is in the knowledge base [25].

Inference rules We can use any of the equivalences from Figure 3 as inference rules, for example:

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad and \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

2.2 Resolution

So far we did not provide any algorithm which can be considered **complete**, since the lack of some *inference rules* may prevent the algorithm from reaching the goal. So we introduce the inference rule called **resolution** that yields a complete inference algorithm when coupled with any complete search algorithm (Section 2.4).

Definitions Following there are some definitions:

- *Formula*: is a set of literals joint by some logic connectives (e.g. a complex sentence).
- *Literals*: can be either one propositional symbol (or atom) or a negated atom. They are the same as an atomic formulae that is a formula that contains no logical connectives.
- *Clause*, is a *disjunction* of literals, for example $L_1 \vee L_2 \vee ... \vee L_n$
- Moreover we introduce the constants $\perp$ and $\top$, that are *False* and *True* respectively.

2.2.1 Conjunctive Normal Form [CNF]

First thing first note that if we have a KB on which we *can* use Modus Ponens *then* we can use Resolution too. On the other hand if we can use Resolution then we may be not able to use Modus Ponent, since generalize Modus Ponens can be only used for definite horn clauses.

Definition The key idea is that: *every sentence of propositional logic is logically equivalent to a conjunction of clauses* ($Formula \equiv CNF(Formula)$) ; hence sentences expressed as a conjunction of clauses are said to be in CNF. You can either preserve equivalence when converting to CNF, but the number of clauses will be 2^n, where n is the number of literals; or you can preserve *satisfiability* introducing new literals and linearly increase the size of the formula.

[25]Nonmonotonic logics, which violate the monotonicity property, capture a common property of human reasoning: changing one's mind.

Example Let's make an example using:

$$A \Leftrightarrow (B \vee C)$$

In the following steps we make use of the formulae in Figure 3

1. Use biconditional elimination and get:
   $(A \Rightarrow (B \vee C)) \wedge ((B \vee C) \Rightarrow A)$
2. Use implication elimination end get :
   $(\neg A \vee B \vee C) \wedge (\neg(B \vee C) \vee A)$
3. CFN requires the negation to be applied only to literals [26], hence we need to use the De Morgan formula to move $\neg$ inwards:
   $(\neg A \vee B \vee C) \wedge ((\neg B \wedge \neg C) \vee A)$
4. Finally we use the distributive law of $\vee$ over $\wedge$ and get:
   $(\neg A \vee B \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A)$

Alpha-Beta Formulas There are some cases in which we need to separate dis/conjunction depending on the following formulas:

C=$\{\alpha\}$	C$_1 = \{\alpha_1\}, C_2 = \{\alpha_2\}$	C=$\{\beta\}$	C=$\{\beta_1, \beta_2\}$
$\alpha = A \wedge B$	$\alpha_1 = A,\ \alpha_2 = B$	$\beta = A \vee B$	$\beta_1 = A,\ \beta_2 = B$
$\alpha = \neg(A \vee B)$	$\alpha_1 = \neg A,\ \alpha_2 = \neg B$	$\beta = A \Rightarrow B$	$\beta_1 = \neg A,\ \beta_2 = B$
$\alpha = \neg(A \Rightarrow B)$	$\alpha_1 = A,\ \alpha_2 = \neg B$	$\beta = \neg(A \wedge B)$	$\beta_1 = \neg A,\ \beta_2 = \neg B$

Table 2: Alpha-Beta formulas for CNF

As you can see from Table 2 the alpha formulas result into two clauses [27] C_1, C_2, while the beta formulas terminate in a single clause C with two formulas [28].

Algorithm Given a formula F the algorithm for having the latter in CNF is the following:

1. Let $I = \{F\}$[29] be the initial state.
2. At the step $n+1$ we have that $I = \{D_1, ..., D_n\}$ where D_i is a disjunction containing some formulas $\{A_1^i, ..., A_k^i\}$.
3. If A_j^i is not a literal then choose D_i and do the following $\forall X_j \in D_i$:
   - if X_j is $\neg\top$ then replace it with $\bot$.
   - if X_j is $\neg\bot$ then replace it with $\top$.
   - if X_j is $\neg\neg X_j$ then replace it with X_j.

[26] When a sentence has negation symbols applied directly to literal then it is in **Negation normal form** [NNF]. For example $\neg(A \vee B)$ is not in NNF while $\neg A \wedge \neg B$ is.

[27] That is because there is a conjunction symbol.

[28] Note that I said formulas and not literals since both $\alpha_1, \alpha_2, \beta_1, \beta_2$ can be further simplified if they are not literals.

[29] The braces indicates that the I is a set of clauses which will be joint by conjunctions.

- if X_j is β formula then replace it with β_1, β_2.
- if X_j is *alpha* formula then replace D_i with two clauses D_i^1, D_i^2 and add them to I.

4. Continue until every D_i is made of literals.

Expansion rules Finally note that the above rules can be written in the form of *expansion rules* [30]:

$$\frac{\neg\neg A}{A} \quad \frac{\neg\top}{\bot} \quad \frac{\neg\bot}{\top} \quad \frac{\beta}{\beta_1, \beta_2} \quad \frac{\alpha}{\alpha_1 | \alpha_2}$$

Another Example Given the formula:

$$(P \Rightarrow (Q \Rightarrow (S \vee T))) \Rightarrow (T \Rightarrow Q)$$

We have that:

1. Use implication elimination (from now on it will be taken for granted) and get:
$$\neg(P \Rightarrow (Q \Rightarrow (S \vee T))) \vee (T \Rightarrow Q)$$

2. Use β rule on disjunction:
$$C = \{\beta_1 = \neg(P \Rightarrow (Q \Rightarrow (S \vee T))), \beta_2 = (T \Rightarrow Q)\}$$

3. Use α rule on β_1 (negation of implication) and get:
$$C_1 = \{P, (T \Rightarrow Q)\}, \quad C_2 = \{\neg(Q \Rightarrow (S \vee T)), (T \Rightarrow Q)\}$$

4. Use β rule on C_1:
$$C_1 = \{P, \neg T, Q\}, \quad C_2 = \{\neg(Q \Rightarrow (S \vee T)), (T \Rightarrow Q)\}$$

5. Use β rule on C_2:
$$C_1 = \{P, \neg T, Q\}, \quad C_2 = \{\neg(Q \Rightarrow (S \vee T)), \neg T, Q\}$$

6. Use α rule on C_2:
$$C_1 = \{P, \neg T, Q\}, \quad C_2 = \{\neg Q, \neg T, Q\} \quad C_3 = \{\neg(S \vee T), \neg T, Q\}$$

7. Use α rule on C_3:
$$C_1 = \{P, \neg T, Q\}, \quad C_2 = \{\neg Q, \neg T, Q\} \quad C_3 = \{\neg S, \neg T, Q\} \quad C_4 = \{\neg T, \neg T, Q\}$$

[30]Using the form introduced in Section 2.1.1.

2.2.2 Horn clause

Definitions The definition for understanding Horn clauses are both reported in Figure 4 as well as in the following list:

- *Definite clause*: is a disjunction of literals of which *exactly one is positive.* For example:
$$(\neg L_1 \lor \neg L_2 \lor L_3)$$
- *Goal clause*: is a disjunction of literals of which *none is positive.* For example:
$$(\neg L_4 \lor \neg L_5 \lor \neg L_6)$$
- *Horn clause*: is a disjunction of literals of which *at most one is positive.* So you can either have *definite clauses* or *goal clauses.* For example:
$$(\neg L_1 \lor \neg L_2 \lor L_3) \quad \textit{or} \quad (\neg L_4 \lor \neg L_5 \lor \neg L_6)$$
- *Horn Form*: a KB is in Horn form if it is a conjunction of Horn clauses. For example:
  1. $C \land (B \Rightarrow A) \land (C \land D \Rightarrow B)$
  2. $C \land (\neg B \lor A) \land (\neg(C \land D) \lor B)$
  3. $C \land (\neg B \lor A) \land (\neg C \lor \neg D \lor B)$

Motivation So why are we interested in definite/goal/Horne clauses?

- When we have a definite clause we can write it as an implication of conjunctions. For example:
$$(\neg L_1 \lor \neg L_2 \lor L_3) \quad \textit{becomes} \quad (L_1 \land L_2) \Rightarrow L_3$$
In this case the premise is called the *body* and the conclusion is called *head.* Moreover a single positive literal is a *fact* [31].
- what is the advantage of having a goal clause??????????
- Moreover inference in horn clauses can be done with the forward/backward chaining algorithm (Section 2.3).
- Finally deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base.

[31] Since a single literal can be viewed as a disjunction of one literal.

$$\boxed{\begin{array}{rcl}
\mathit{CNFSentence} & \rightarrow & \mathit{Clause}_1 \wedge \cdots \wedge \mathit{Clause}_n \\
\mathit{Clause} & \rightarrow & \mathit{Literal}_1 \vee \cdots \vee \mathit{Literal}_m \\
\mathit{Literal} & \rightarrow & \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} & \rightarrow & P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} & \rightarrow & \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} & \rightarrow & (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} & \rightarrow & (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{array}}$$

Figure 4: Different types of clauses

2.2.3 Propositional Resolution

The key idea of the propositional resolution is that if I have two clauses C_1, C_2 [32] and one literal L that appears with different sign in both $L \in C_1$, $\neg L \in C_2$, then I can generate a new clause by joining $C_1 \vee C_2$ and removing $L, \neg L$ from them. That is because if I have a clause where a symbol appears with both sign, e.g. $C_3 = L_1 \vee L_2 \vee \neg L_1$, then every interpretation I can give to L_1 will be indifferent since it will result in a model.

English Example Consider the following KB, with two clauses, trying to understand why can't I play Overwatch on Monday:

- C_1=*"There is an exam the next day"* or *"My pc is broken"* or *"I'm tired"*
- C_2=*"There is no exam the next day"*

This is a special case of resolution called **unit resolution** in which I have a clause C_1 and a fact C_2. So, by examine the two clauses I can say that *There is no exam the next day*, so the alternatives are either that *my pc is broken* or that *I'm tired.*
If we add another fact to the KB:

- C_3= *I'm never tired on Mondays!*

Then the only logical conclusion why I can't play Overwatch on Monday is because *my pc is broken.* [33].

Formalization We can formalize the previous example for any given set of clauses:
Let there be two clauses:

$$L = L_1 \vee L_2 \vee ... \vee L_n = \{L_1, ..., L_n\} \tag{1}$$
$$P = P_1 \vee P_2 \vee ... \vee P_m = \{P_1, ..., P_m\} \tag{2}$$

[32]You should know by now that a clause is a disjunction of literals.
[33]Sad me.

for any n, m.
Let there be a set of literals, $O = \{O_1, ..., O_k\}$, which appears in both L and P but with opposite sign:

$$\forall O_i,\ O_i \in L, \neg O_i \in P$$

Se we can use resolution to join P and L [34] and remove O from the union:

$$\frac{L \quad P}{(L \cup P) \setminus O}$$

Which result will look like:

$$(L_1 \vee L_2 \vee ... \vee L_n \vee P_1 \vee P_2 \vee ... \vee P_m) \setminus (O_1 \vee ... \vee O_k)$$

Formal Example Let's use the following KB:

$$KB = \{C_1 = \{\neg P, Q, \neg P\}, \quad C_2 = \{P, \neg L\}\}$$

We first apply **factoring** to remove the double $\neg P$ in C_1, thus having:

$$KB = \{C_1 = \{\neg P, Q\}, \quad C_2 = \{P, \neg L\}\}$$

Given a formula $\alpha = \{Q, \neg L\}$ we want to know if $KB \models \alpha$, that is $(KB \vee \neg\alpha)$ is unsatisfiable, that is $(KB \vee \neg\alpha) \vdash_{\mathcal{R}} \{\}$ [35]. We have that:

$$(KB \vee \neg\alpha) = \{C_1 = \{\neg P, Q\}, \quad C_2 = \{P, \neg L\}, \quad C_3 = \{\neg Q\}, \quad C_4 = \{\neg\neg L = L\}\}$$

The procedure works as follow:

1. Resolve C_1 with C_2 which outputs:

$$C_5 = \frac{\{\neg P, Q\} \quad \{P, \neg L\}}{\{\neg L, Q\}} = \{\neg L, Q\}$$

   So that we have:

$$C_3 = \{\neg Q\}, \quad C_4 = \{L\}, \quad C_5 = \{\neg L, Q\}$$

2. Resolve $C3$ and C_5 in the same way and get:

$$C_4 = \{L\}, \quad C_6 = \{\neg L\}$$

3. Finally resolve C_4 and C_6 and get the empty clause $\{\}$

We just demonstrated that $(KB \vee \neg\alpha) \vdash_{\mathcal{R}} \{\}$ thus that $KB \models \alpha$. Notice that these passages can be written in a tree form as shown in Figure 5.

[34] In this instance by joining we mean that we use the disjunction symbol between the two clauses.
[35] Note that $\{\}$ is the empty clause which is equivalent to *False*.

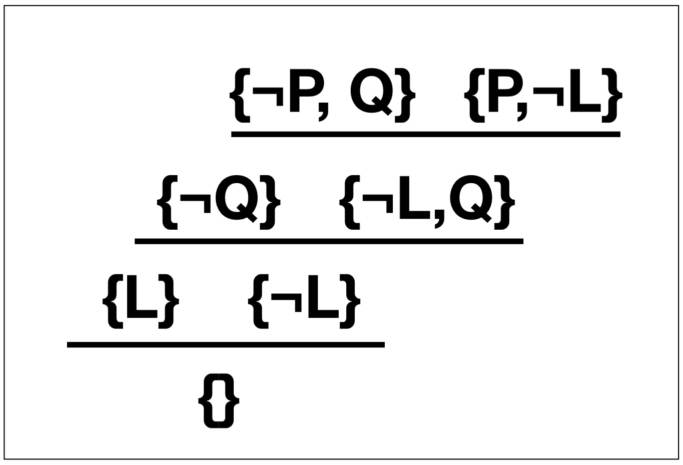


Figure 5: Resolution steps shown in tree form

Satisfiability of Resolution We start by having two clauses $C_1 = \{L_1, ..., L_n\}$, $C_2 = \{P_1, ..., P_m\}$ and a literal that is the same in each clause but with a different sign $L_i = \neg P_j = K$, i.e. if L_i is *True* then P_j is *False*. We want to prove that $\{C_1 \cup C_2\} \setminus \{K\}$ is satisfiable.
Suppose C_1 and C_2 are satisfiable, i.e. there exists a model $\mathcal{M}$; such that $\mathcal{M} \models C_1$ and $\mathcal{M} \models C_2$, and that L_i is true in $\mathcal{M}$; it follows that $\neg L_1$ is *False* in $\mathcal{M}$, thus P_j is *True* in C_2 and therefore C_2 must be *True* in $\mathcal{M}$. Consequently, $C_1 \cup C_2$ is true in $\mathcal{M}$. Since $\mathcal{M}$ is a model of C_1, C_2 by hypothesis, $C_1 \cup C_2$ is true in M.
Same goes with $\mathcal{M} \models \neg L_i$.

Completeness of resolution We introduce the notion of **resolution closure** $RC(S)$ of a set of clauses S that is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives. It is easy to understand that $RC(S)$ is finite because there are only finitely many distinct clauses that can be constructed out of the symbols $P_1, ..., P_k$ that appear in S [36].

Now let us consider the **ground resolution theorem** which states: *If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause*; that is :

$$S \quad unsatisfiable \quad iff \quad \{\} \in RC(S)$$

Now let's prove this the other way around, we want to prove that:

$$S \quad satisfiable \quad iff \quad \{\} \notin RC(S)$$

First we build a model of $RC(S)$ with the following procedure.
For i in $range(k)$:

[36]This is true only if we are using the factoring step to remove duplicate literals.

- Given a clause $C_k \in S$ that contains $\neg P_i \in C$ and all its other literals are set to *False*, $C_k = false \vee ... \vee false \vee \neg P_i$; assign $False$ to P_i, so to make C_k true.
- Otherwise assign *True* to P_i.

So now we have a model of S. Since we must prove that $RC(S)$ is satisfiable, thus have some models, let's assume that what we just got is not a model, then we must have been wrong at some iteration i. That is setting the symbol P_i causes some clause C to becomes false, since we *do not have a model* all the other clauses must be already false. So C can be one of the following possibilities:

- $C = false \vee ... \vee false \vee P_i$
- $C = false \vee ... \vee false \vee \neg P_i$

If *only one* of the two is in $RC(S)$ then the algorithm would assign the right value to make C true. So the only way to have C false is that *both* possibilities are in $RC(S)$, but this is impossible since $RC(S)$ is **closed under resolution**, that is the two possibilities would have been resolved by the algorithm.

2.3 Chaining

2.3.1 Forward Chaining

Definition Determines if a single proposition symbol q [37] is entailed by a KB of definite clauses [38]. It begins from known facts in the knowledge base [39]. If all the premises of an implication are known, then its conclusion is added to the set of known facts. This process continues until the query q is reached or until no further inferences can be made. The main point to remember is that it runs in linear time.

Example Given the following KB:

- $P \Rightarrow Q$, equal to $\neg P \vee Q$ (definite clause)
- $L \wedge M \Rightarrow P$, equal to $\neg L \vee \neg M \vee P$ (definite clause)
- $B \wedge L \Rightarrow M$, equal to $\neg B \vee \neg L \vee M$ (definite clause)
- $A \wedge P \Rightarrow L$, equal to $\neg A \vee \neg P \vee L$ (definite clause)
- $A \wedge B \Rightarrow L$, equal to $\neg A \vee \neg B \vee L$ (definite clause)
- A, (known fact, since positive literal)
- B, (known fact, since positive literal)

We query Q [40].
The KB can bee seen as an **AND-OR Graph** where the implications are *or* arches while the $\wedge$ are *and* arches (Figure 6).

[37] What is asked to the KB, i.e. the query.
[38] Definite clauses means the KB contains either facts (positive literals) or implication with an *atomic* conclusion and, for premise, either an atom or a conjunction of literals.
[39] For this reason it is called **data driven**.
[40] That is we ask the KB about the truthfulness of Q.

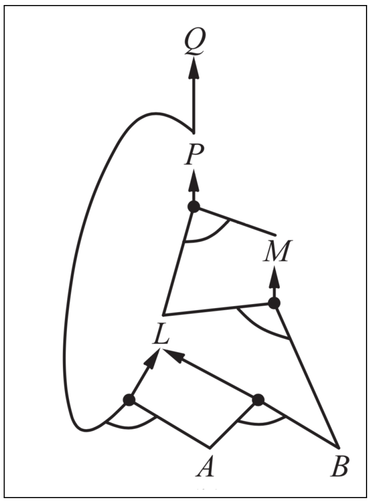


Figure 6: AND-OR graph for definite clause KB

The flow of the algorithm works as illustrated in Figure 7.

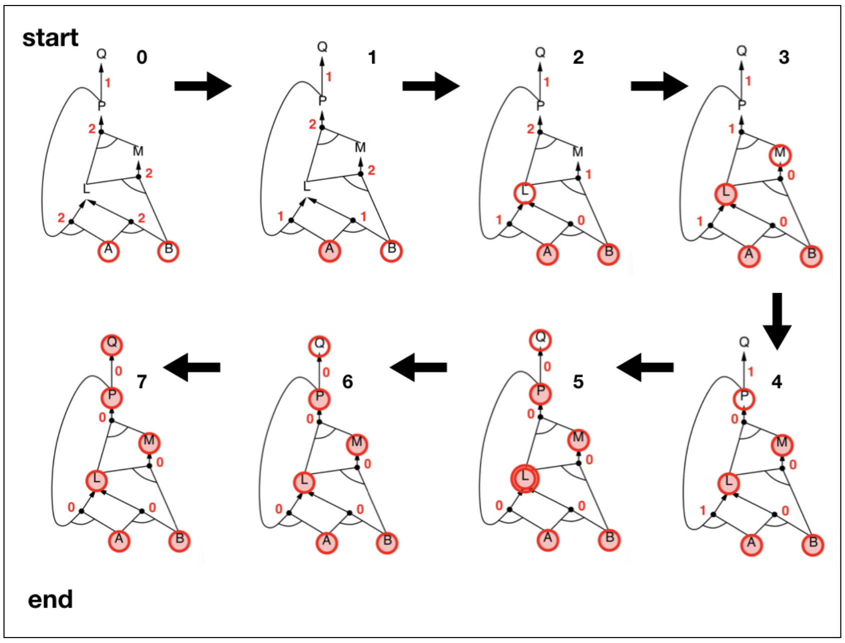


Figure 7: Forward Chaining for KB graph

Proof We can affirm that the algorithm is **sound** since every step is an application of Modus Ponens.
For the **completeness** we need to prove that every possible atomic sentence can be derived from the KB. We first introduce the notion of **fixed point** which is a state of the algorithm in which no more inference can be done; then we

consider to be in the final state m that is a model [41] because we cannot apply Modus Ponens to any other clause.
Since the final state is in Horn form [42] we have a conjunction of disjunctions, so to be a model [43] every clause must be *True.* To prove this let's consider the opposite:

- Let's take a generic clause $C = a_1 \wedge ... \wedge a_n \Rightarrow b$ in m. This clause have a general number of conjunctions n in its premise.
- Suppose that C is false in m.
- For an implication to be *False* we need the premise $a_1 \wedge ... \wedge a_n$ to be *True* and the conclusion b to be *False.*
- But if that is the case I would apply Modus Ponens again $\frac{a_1 \wedge ... \wedge a_n \Rightarrow b \quad a_1 \wedge ... \wedge a_n}{b}$ and conclude b True, so in fact I'm not in a fixed point and this contradicts the assumption.

We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB. Furthermore, any atomic sentence q that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence q must be inferred by the algorithm.

2.3.2 Backward Chaining

On the other hand the backward chaining works its way from the query q [44] and find those implication which conclusions are equal to q. If all the premises of a given implication which resolve in q are true then we can conclude that q is true itself.
This kind of reasoning is called **goal directed** reasoning and it usually works in linear size in respect to the size of the KB.

2.4 Proposal for model checking

In this section, we describe two families of efficient algorithms for general propositional inference based on model checking: One approach based on backtracking search, and one on local hill-climbing search. Notice that these algorithms are used for checking the **satisfiability** (SAT) of a problem

2.4.1 DPPL

Also known as David-Putnam, Logemann, Loveland algorithm, it takes as input a sentence in conjunctive normal form (Section 2.2.1) and uses a recursive depth-first enumeration of all possible interpretations. To speed up the algorithm we can introduce some improvements.

[41] In which there has been various assignment of *True/False* for the literals.
[42] As the whole KB.
[43] True interpretation.
[44] Note that if the query is true the algorithm stops immediately.

Early Termination A clause is true if *any* literal is true [45], hence if we encounter a true literal in a clause we can stop looking at it and give it the value *true* [46].
For example having $(A \vee B) \wedge (A \vee C)$, if A is true then we do not need to look for the values of B, C.

Pure Symbol is a symbol which always appears with the same ”sign”, i.e. negated or not, so it can be assigned a value once for all the clauses. For example:

$$(A \vee \neg B) \wedge (\neg B \vee \neg C) \wedge (C \vee A)$$

In this case the literal A is a pure positive symbol, B is a pure negative while C is impure.

Unit clause is a clause with only one literal. This can be either because

- We have a clause with just one literal (fact). Or
- We have a clause with multiple literals that are false [47] except for this last one [48].

Component analysis We can divide clauses into independent subset when they *do not share* any common literal. By dividing them we can parallelize the job as well as prune large part of the state space [49]

Variable and Value ordering A general rule is to always try the value *true* before *false*. While the **degree heuristic** suggests choosing the variable that appears most frequently over all remaining clauses [50].

Intelligent Backtracking Also discussed in the Planning part of the course, intelligent backtracking keeps tracks of conflicts [51] so to cut off useless searches steps.

Random restarts When the algorithm gets stuck for a long time execute a restart from the root point.

Clever indexing at implementation level.

2.4.2 Local Search Algorithm

These algorithms works by flipping the truth value of one symbol at a time. The search space usually contains many local minima, to escape from which various forms of randomness are required.

[45] Remember that a clause is a disjunction of literals of the form $C_1 \vee C_2 \vee ... \vee C_n$.
[46] Similarly a sentence (conjunction of clauses) is false if *any* clause is false.
[47] Since clauses are disjunction, false literals can be removed if there are some other that can assume the value true.
[48] This is also called **unit propagation**.
[49] No necessity to check the constraint of a literal in other clauses which do not have it.
[50] Everything we saw in the planning part of the course.
[51] Literals that may create a conflict with other literals.

GSAT Missing

WALKSAT The algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip:

- A *min-conflicts* step that minimizes the number of unsatisfied clauses in the new state.
- A *random walk* step that picks the symbol randomly.

This type of algorithm is very similar to simulate annealing studied in the Local search part. WALKSAT may end with the following outcomes:

- A model; thus the input sentence is satisfiable.
- A failure; then there are two possibilities:
  - Either the sentence is unsatisfiable, or
  - The algorithm needs more time to complete the search [52].

2.4.3 Random SAT problem

Depending on the number of clauses m and symbols n we can either have an *under-constraint* problem ($n > m$) or a *over-constraint* problem ($m > n$) [53]. Given the ration $r = m/n$, when the ratio approaches zero then the probability of the problem to be satisfiable approaches one and vice versa. As you can see from Figure 8 around the value $r = m/n = 4.3$ the probability for the sentence to be satisfiable approaches zero.

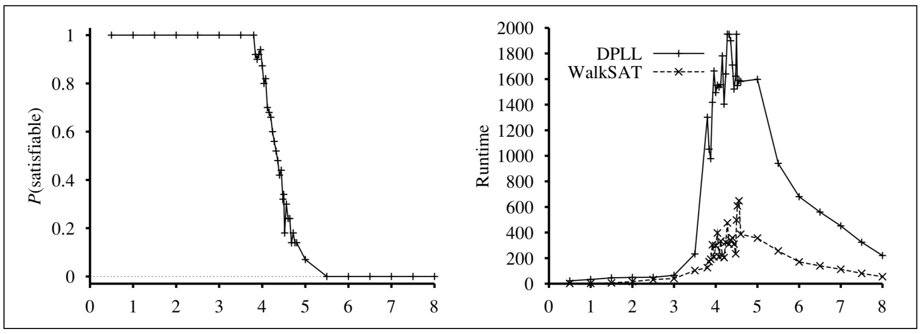


Figure 8: (Left) Graph showing the probability that a random sentence with n = 50 symbols is satisfiable, as a function ratio m/n. (Right) Graph of the median run time on random sentences. The most difficult problems have a ratio of about 4.3.

[52]Note that if there is no limit to the number of flips and the sentence is unsatisfiable then the algorithm may never end.

[53]Giving that each symbol may not appear twice in a clause and no clauses may appear twice in the sentence.

3 First Order logic

One of the most important proprieties a bout propositional logic that holds for First Order Logic (FOL from now on) is the **compositionality**, that is the meaning of a sentence is a function of the meaning of its parts. For example given $S_1 =$*I have don't want to write this* and $S_2 =$*I would love to watch Netflix* if would be weird if $S_1 \wedge S_2 =$*Tomorrow will be cold.*
Moreover propositional logic assumes that there are facts that either hold or do not hold in the world [54], while FOL assumes that the world consists of objects with certain relations among them that do or do not hold. Hence in both logic a sentence represents a fact and the agent either believes the sentence to be true, believes it to be false, or has no opinion.

Structure FOL assumes the world contains:

- **Objects** like: pc, tv, beer....
- **Relations** is a set of tuples of objects that are related. Depending on the number of tuple in the set we can have:
  - *Proprieties* (or unary relationship) that have just one tuple in the set; such as: red, round, big...
  - *General* (or n-ary relationship) which may have n tuples; like: bigger than, has color, owns...
- **Functions** For example: father of, best friend...

Example Given the sentence *"The next MidTerm will be easier then the first one"*[55] I have that:

- *Objects*: Midterm, one
- *Proprieties*: next, first (unary)
- *Functions*: easier than

Symbols For each one on the structures we have a related symbol:

- **Constants**: stands for objects.
- **Predicates**: stands for relations.
- **Functions**: stands for functions [56]

The last two comes with an **arity** that is the number of arguments they are referring to [57].

[54]Each fact can be in one of two states: true or false.
[55]Probably not a model.
[56]You don't say.
[57]For example the function *Sibling(x,y)* has arity 2 (since we need a subject and a brother), while the function *Human(x)* has arity 1.

Interpretation vs Model A model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects.

Terms are a logical expressions that refers to an object. That is what is used to refer to a particular object in the domain; they can be:

- **Constants** symbols; for example the color red, or that guy Eric over there.
- **Functional** terms; which are functions applied to constant symbols such as: *BestFriendOf(Eric)* or *Mother(Marta)*.

Notice that both refer to an "object" in the domain, one directly (constants), while the other indirectly. A functional term can be used both for not known constants, like the name of Marta's mother, but they can also be used for thing we do not bother to name, e.g. *LeftArmOf(Edoardo)*.

Atomic Formulas An atomic formula (or atom for short) is a predicate symbol with a number $n \in [0,]$ of terms. For example given $t_1, ..., t_n$ terms, the following are all atoms:

- *Sibling*(t_i, t_j): predicate symbol with arity 2.
- $Friend(t_1, ..., t_n$: predicate symbol with arity n.
- $t_i = t_j$: predicate symbol = with arity 2, can be written as $Equals(t_i, t_j)$
- $ParentOf(ParentOf(x))$ is the grandparent of x and has arity 1.
- $\bot, \top$ are atoms.

Terms vs Atoms So what is the difference between this two symbols?
The difference is in the use of relationships, for example:

- *BatCave, CaveOf(Batman)* are terms, while *Big(BatCave),Big(CaveOf(Batman))* are atoms.
- Equal symbols generate atoms, for example *MobileOf(Batman)* and *BatMobile* are terms, where *MobileOf(Batman)=BatMobile* is an atom.

Complex Sentences When we have atoms joint by some logic connectives then we call them **complex sentence**, for example:

- $\neg Friends(Superman, LexLutor)$
- $\wedge Kryponite \Rightarrow Weak(Superman)$
-