Course: 231 Introduction to AI

Knowledge and Inference

- Recall basic concepts of logic
- Logical inference
  - deduction
  - *abduction*
  - *induction*
- Clausal Logic
- Deductive Inference (e.g. resolution)
- Recap of SLD and SLDNF

Logic: another thing that penguins aren't very good at.

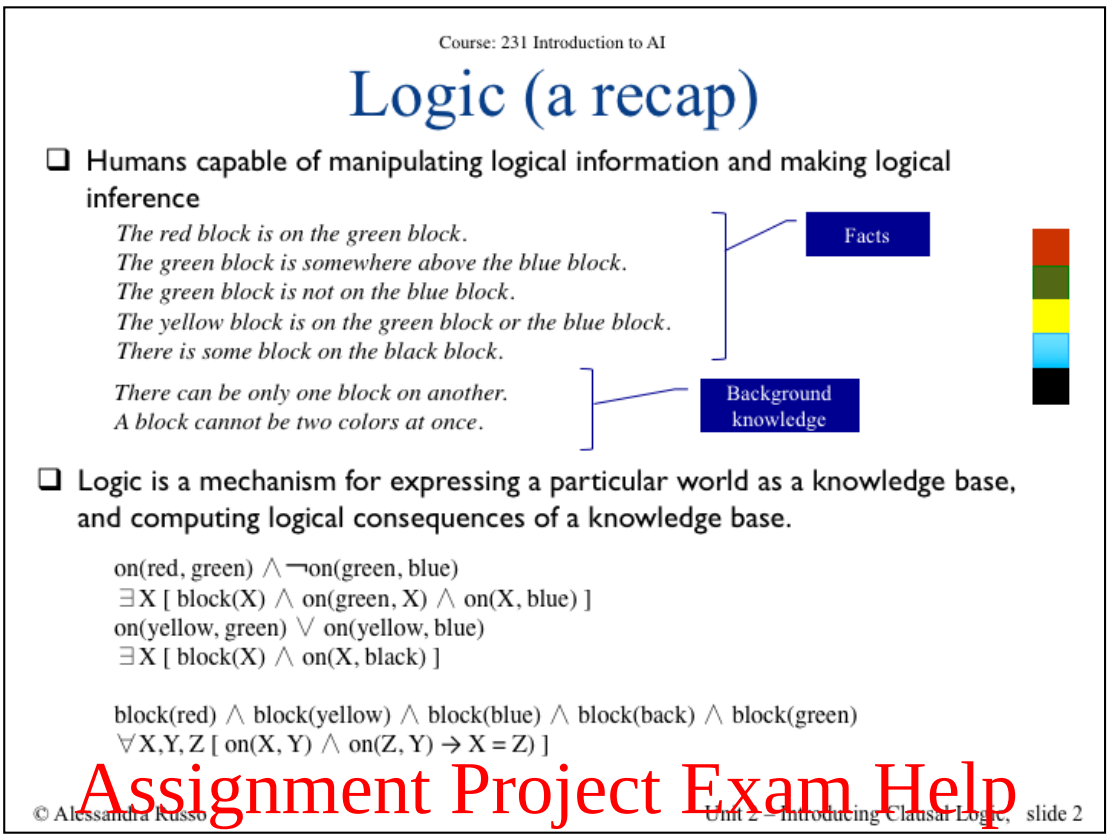
In this unit, we will recap key concepts for modelling knowledge in computational agents. Our modelling will have to be able to express and support reasoning about objects in the real world and relations between objects. So the level of expressivity that we require is that of predicate logic. But at the same time it has to be computationally feasible. We will recall basic notions of predicate logic , which will relate to your first year Logic course, and we will see restrictions that we will need to apply to this form of representation to make the inference computationally tractable. When we will talk about SAT solving, we will relax some of these restrictions, refer back to full classical logic, but considering propositional representations in order to maintain the decidability of the computational problem in hand.

So, we will quickly recall basic concepts of predicate logic. We will present three forms of knowledge inference, known as *deduction*, *abduction* and *induction* and briefly illustrate how they differ from each other.

We will then focus on computational aspects of reasoning about individuals and relations. We will start with summarising the concept of clausal logic and resolution for clausal logic, which is one of the main proof methods used for computing consequences from a given knowledge base represented in clausal form. We will then briefly summarise Horn clauses, a subset of clausal logic that is at the core of declarative programming, and highlight the advantage of reducing the expressivity of the language in order to gain efficiency in the computational mechanism (e.g. SLD-resolution). You have already covered the semantics of this form of knowledge representation in the first part of this course. So we will concentrate on a recap on the inference process: give some examples of SLD-resolution and show how it is extended to handle deductive inferences in the context of normal clauses. These are clauses that include negation by failure. This unit is essentially a brief summary of some of the topics that have already been covered in the first part of this course, but I will introduce the notation that I will be using.

Next week we will be focusing on the notion of abductive reasoning and its applications. We will define one of the algorithms for computing explanations, formalise its semantics, discuss the role of constraints in solving an abductive inference task and give various examples of abductive derivations.

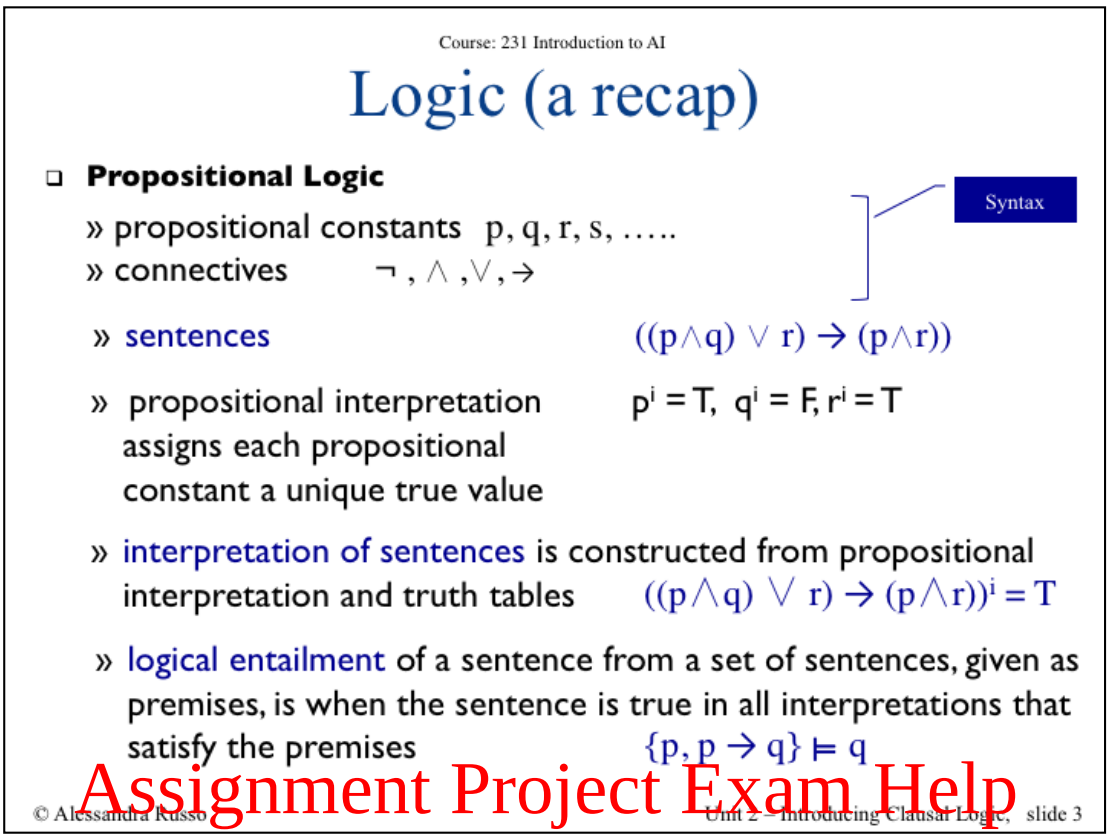
Your first tutorial will be based on constructing deductive proofs for problems that are formalised using the various types of predicate logic formalisms presented here.



People have the ability of acquiring information about the world and use this information to further derive knowledge. They can for instance represent, manipulate logical information, including not just simple facts but also more complex forms of information, such as negations, alternatives, constraints, etc. For instance, given the above set of premises (e.g. some facts) about the arrangement of five blocks in a tower (in a situation where there are only these five blocks) people are capable of inferring the exact configuration of the tower. To compute consequences of a given knowledge base, humans mentally construct *proofs*: sequence of intermediate steps where at each step straightforward intermediate consequences are computed.

Logic is a mechanism for representing a particular world that the human has in mind, using symbols and for computing consequences (i.e. valid logical inferences). The symbols used in a logical formalism are purely syntactic. A computational agent does not know the meaning of these symbols. But their denotational semantics, or intended interpretation, is the particular real world that the human has in mind. Consequences are computed through just pure syntactic manipulation of the expressions in the knowledge based. The human knows how to interpret the answers with respect to the intended interpretation, that is the particular world. Because the intended interpretation is a model of the knowledge based, and a logical consequence is true in all models, a logical consequence must be true in the intended interpretation.

Logic is therefore defined in terms of a syntax (language), a formal semantics and a set of rules that help manipulate mechanically given information (or knowledge). Above is an example of formal representation in predicate logic of the given facts and background knowledge. Applying, for instance, proof procedures such as resolution or natural deduction, it would be possible to derive the exact configuration of the tower as conjunction of five ground atoms. We will give later in this unit, a particular representation style of predicate logic, called “clausal logic”, for which the inference mechanism called resolution (closely related to Prolog) can be used to compute answers to given queries.

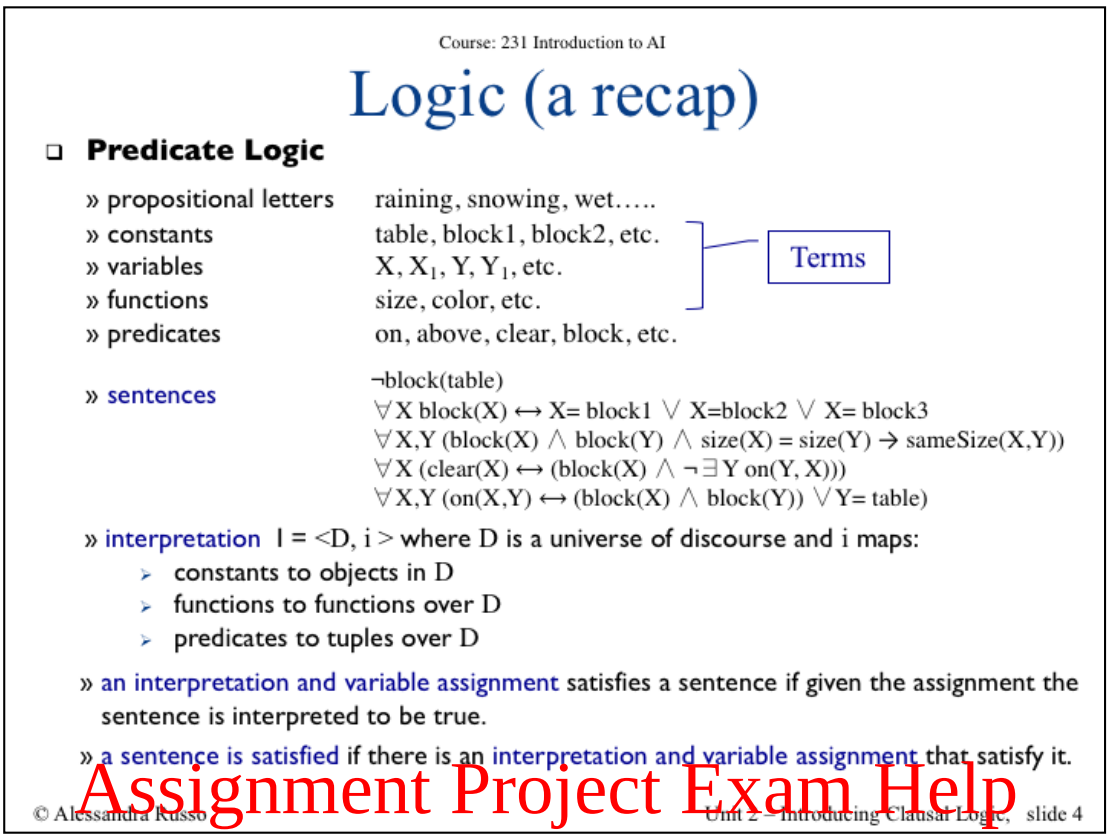


Propositional Logic is concerned with propositions and their logical interrelationships. A proposition is normally a possible "feature" of the world about which we want to say something. This feature needs not be true for us to talk and reason about it. As any other logic, propositional logic consists of a syntax and a semantics. The former defines the language with which we can express properties about the world; whereas the semantics defines the meaning of sentences written in this language.

The syntax of propositional logic is given by the main *logical connectives* $\wedge$, $\vee$, $\neg$, $\rightarrow$ and *extra-logical symbols*, also called propositional constants, that form the signature of the language chosen to represent the world. For convention, propositional constants are written using sequence of alphanumeric characters beginning with a lower case character. Sentences are propositional constants or expressions inductively defined in terms of simpler sentences composed together via the logical connectives. For instance $((p \wedge q) \vee r) \rightarrow (p \wedge r))$ is an implication sentence formed by the two simpler sentences $(p \wedge q) \vee r)$ and $(p \wedge r)$ linked together by means of the implication ($\rightarrow$) connective. Remember connectives have a fixed operator precedence and if sentences are constructed without the use of parenthesis the predefined precedence applies.

The semantics of propositional logic is concerned with logical meaning of sentences. This is determined by the logical meaning of the connectives, which is fixed and given in terms of truth tables, and the logical meaning of the propositional constants. The latter is not fixed and it is defined in terms of an *interpretation*: assignment of a truth value (Boolean meaning) to each propositional constant in the language. We will denote an interpretation using the letter *i* and the meaning of a propositional constant or (complex) expression under an interpretation *i* by superscripting the constant or expression with *i*.

An interpretation that makes a given sentence true is said to *satisfy* the sentence. A sentence is *unsatisfiable* if and only if it is not satisfied by any interpretation. A sentence is said to be *valid* if and only if it is satisfied by every possible interpretation. Given a set S of sentences, a model of S is an interpretation that satisfies all the sentences in S. Finally, key to the logic is the notion of logical entailment. A set S of sentences logically entails a sentence φ (written $S \models \varphi$) if and only if every interpretation that satisfies S also satisfies φ.

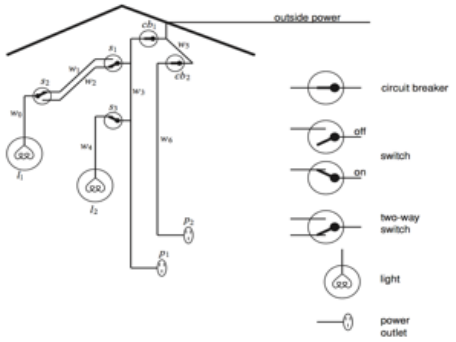


Propositional Logic does a good job of allowing us to talk about relationships among propositions, and it gives us the machinery to derive logical conclusions based on these relationships. Unfortunately, when we want to say general properties, as for instance "*if one person knows a second person, then the second person knows the first*" , we find that Propositional Logic is inadequate. Predicate Logic solves this problem by providing a finer grain of representation. In particular, it provides us with a way of talking about *individual objects* and their *relationships*. It allows us to assert the existence of objects with a given relationship and allows us to talk about all/some objects having a given relationship and to apply those facts to specific cases. The formal language extends that of propositional logic with two additional logical operators, quantifiers $\forall$ and $\exists$, and extra-logical symbols that include constants, function and predicates. The formal language is indeed more expressive that propositional logic in the sense that it allows us to talk about elements of a given domain of discourse, using constant symbols; it allows us to specify functions over this domain, by means of function symbols, and mainly it allows us to define and specify relations among elements of the domain, by means of predicate symbols. By convention, variables are assumed to be written as sequences of alphanumeric characters beginning with a capital case character whereas function and predicates to begin with a small case character. Terms in the languages are constants, variables and any term constructed by applying a function to term arguments.

Logical formulae are negations, conjunctions, disjunctions, implications, and equivalences constructed exactly as in propositional logic, except that the elementary components are most of the time relational formulae rather than propositional constants. *Quantified sentence*s are logical formulae formed from a quantifier, a variable, and an embedded sentence. The embedded sentence is called the scope of the quantifier. There are two types of quantified sentences: universally quantified sentences and existentially quantified sentences. Each variable that appears in a sentence is quantified. A sentence is *ground* if and only if it contains no variables. For example, the sentence *block(table)* is ground, whereas the sentence $\forall X.block(X)$ is not. An interpretation is given by a domain of discourse and a mapping that assigns constant symbols to elements of the domain, function symbols to functions with the same arity over the domain and each predicate to a relation with the same arity over the domain. Satisfiability depends on a given interpretation and variable assignment.

Course: 231 Introduction to AI

Example: Electric Environment



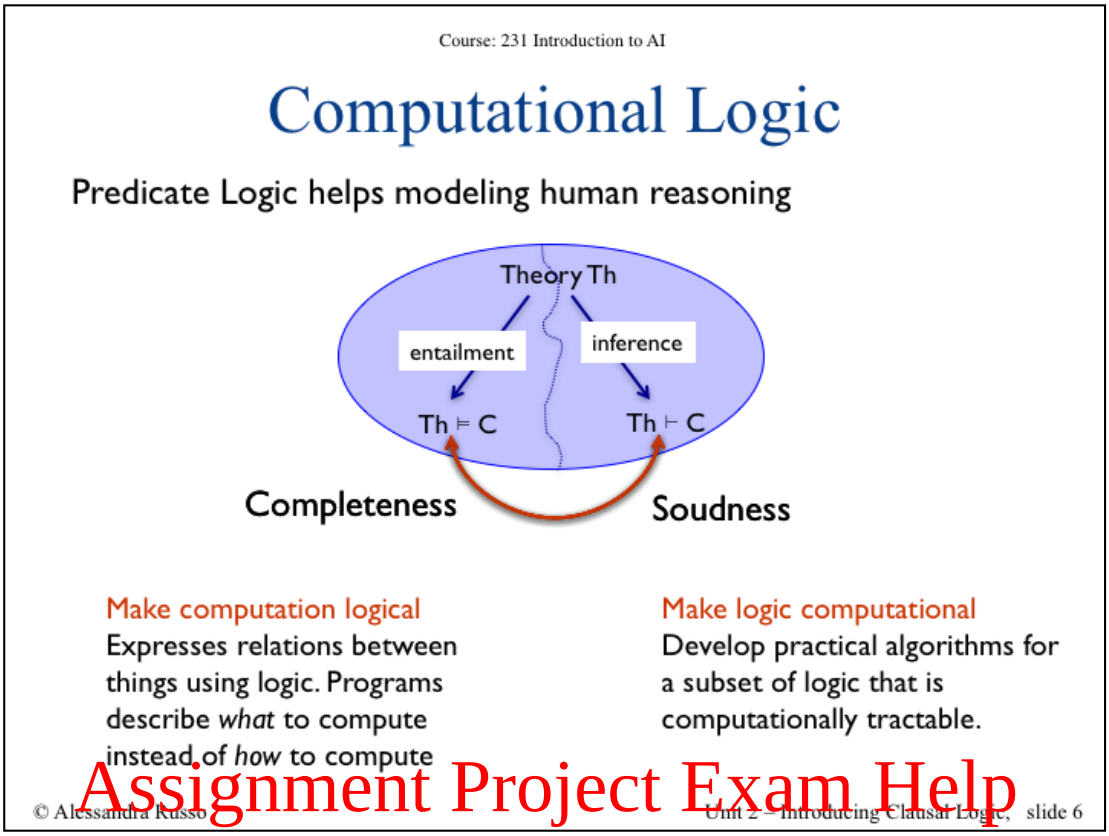
light(l1).
light(l2).
down(s1).
up(s2).
up(s3).
ok(cb1).
ok(outside).
connectedTo(l1, w0).
connectedTo(l2,w4).
live(outside).
connectedTo(w0,w1) ← up(s2).
connectedTo(w0,w2) ← down(s2).
connectedTo(w1,w3) ←up(s1).
connectedTo(w4,w3) ←up(s3).
connectedTo(w3, w5) ← ok(cb1).
connectedTo(w5, outside) ←ok(outside).
lit(L) ←light(L), live(L), ok(L).
live(X) ←connectedTo(X,Y), live(Y).

Query: ? lit(L)

In this slide we give a simple example of clausal representation of a real world scenario of an electric environment. The extensional knowledge base is shown in green, the intentional knowledge base is shown in red. Given this knowledge a monitoring agent would be able to answer which light is on in the building. The query is indicated in blue, and asks "what is lit up in the building".

Use this example to check if you know what the signature of the language is in this case, what predicates, constants and variables are. What is then the answer? Do you know how it is computed?

We will summarise in the next few slides how we can compute it.

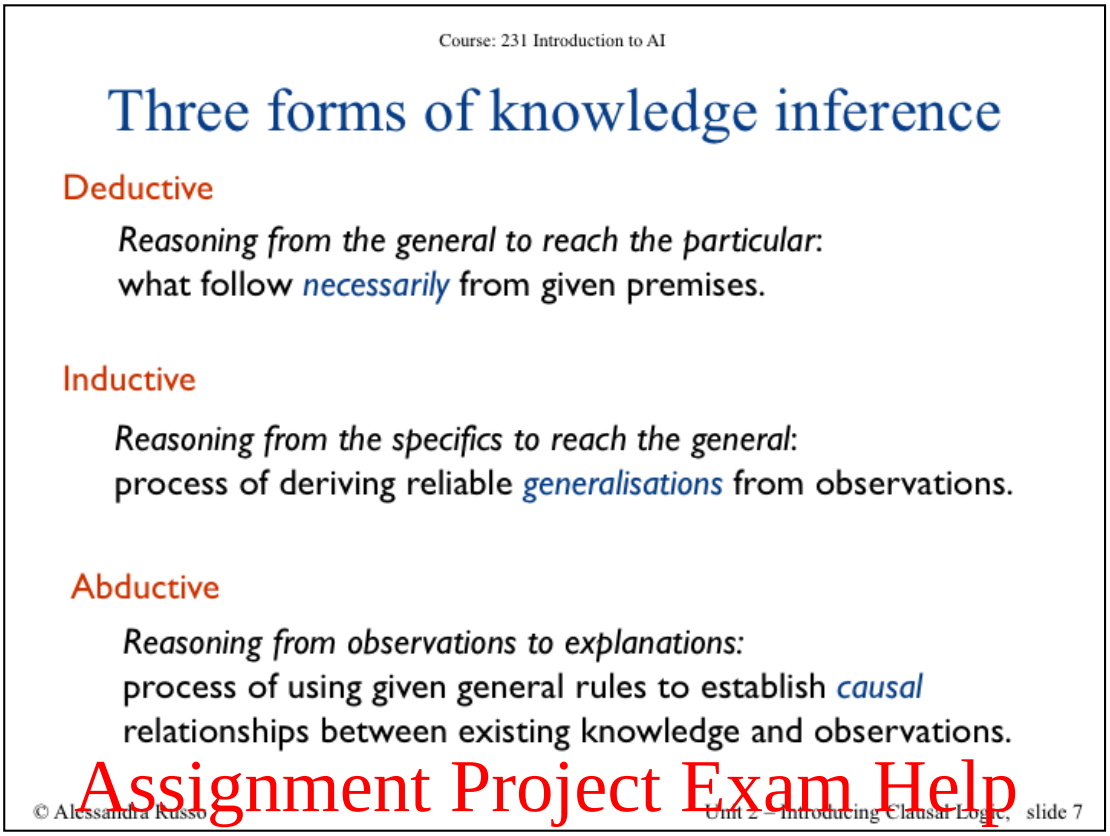


Logic is, however, not only a mechanism for modeling the real-world. The semantics provides the mathematical framework for formally studying validity and correctness of the logical model. But its syntax and proof procedures provide also ways for syntactically manipulating sentences and infer new knowledge from given assumptions or knowledge bases.

Traditional AI, nowadays called also explainable AI, is based on Computational Logic. This is the area of logic that focuses on the development of declarative programming mechanisms, such as logic programming, which make "computations logical". But, at the same time, it investigates algorithms capable of computing inference in a practical way so to return answers to posed questions. In Logic Programming, for instance, a program consists of a set of axioms (facts or extensional knowledge) and a set of rules (also called intentional knowledge). They define relations between entities in a logical manner. Logic programming systems, such as Prolog, compute the consequences of axioms and rules to answer queries.

Part of Computational Logic is also Automated Theorem Proving that studies tractable algorithms for computing inferences. Deductive databases, for instance, use Datalog (a subset of predicate logic) to model relational databases and queries expressed in Datalog can be computed using SQL. Rules defining new predicates can instead be computed using *views* of intentional relations. All these areas can be seen as belonging to computational logic.

The two aspects of computational logic (i.e. defining a problem in logic term and providing algorithms for solving inference in a practical way) are typical of various types of logic-based reasoning. Depending on the problem, different logic-based computations maybe required. Going back to few of the greatest philosophers in the 20th century (e.g. Karl Popper and subsequently Pierce), there are three different forms of reasoning: *deduction*, *abduction* and *induction*. Computational problems formalised in a logical manner can refer to any one of these three forms. In the first case the problem is referred to as a *deductive task*, in the second case as an *abductive task* and in the third case an *inductive task*. On the other hand different computational algorithms have been developed to support the computation of these tasks. We will illustrate the difference between these forms of reasoning, by presenting first deduction, then defining abduction and then focusing on induction. We will look at them as computational tasks. So we concentrate on subset of predicate logic that is computationally tractable (i.e. Horn clauses).

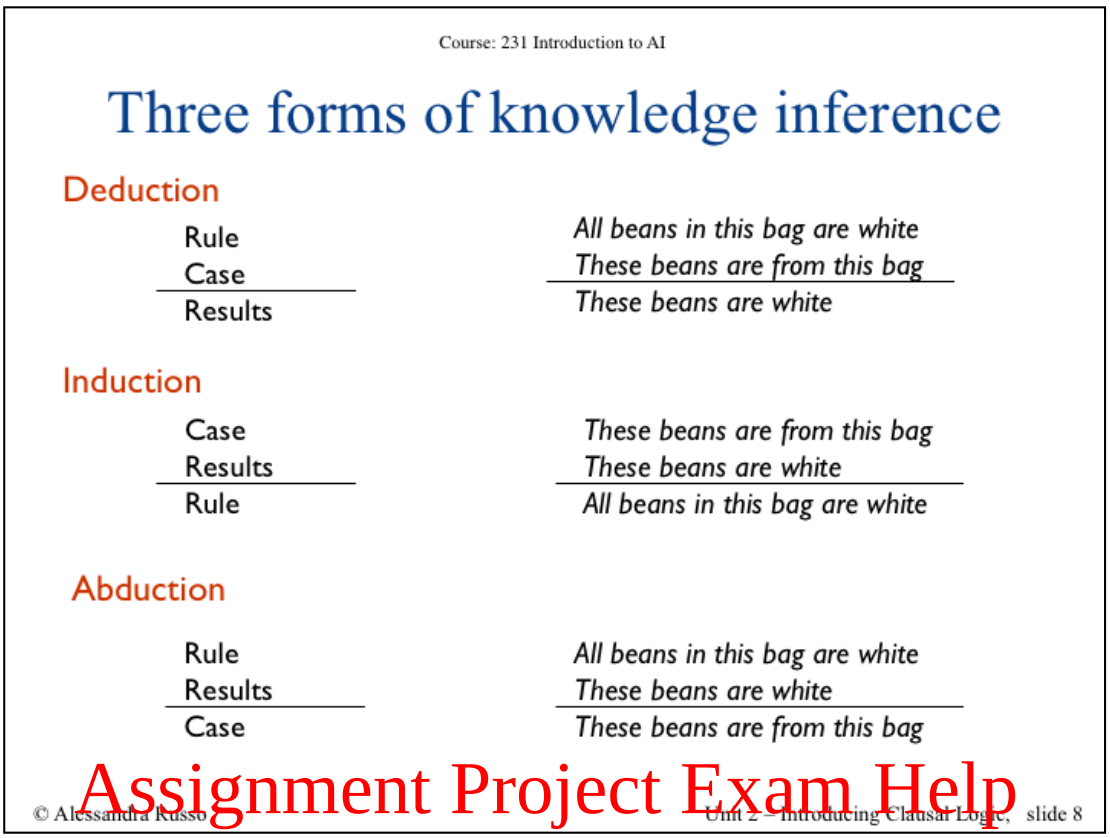


Abduction and induction are two forms of knowledge inference that alongside deduction have played a prominent role in Artificial Intelligence and Philosophy of science. Historically, these three types of reasoning have their roots in the work of Aristotle. But were placed in their modern context of logic-based inference by C.S. Peirce around the twentieth century. The distinction between deduction, on the one hand, and induction and abduction, on the other hand, corresponds to the distinction between *necessary and non-necessary inference*. In deductive inferences, what is inferred is *necessarily true* if the premises that it is inferred from are true; that is, the truth of the premises guarantees the truth of the conclusion. But not all inferences are of this type. Abductive and inductive inference are non-necessary inferences. They are instead *ampliative inferences*.

The key distinction is that whereas deduction aims to make explicit the consequences already implicit in some existing knowledge base, abduction and induction aim to discover new knowledge from empirical data, or observations, about a phenomena.

Abduction is the process of *explanation* — reasoning from effects to possible causes; while induction is the process of *generalisation* — reasoning from samples to wider populations. Significant progress has been made in the areas of Artificial Intelligence (AI) and Machine Learning (ML), on formalising and automating these forms of reasoning. This progress has led to deeper understanding and increasing real-world applications. Much of this success comes from work in the areas of Abductive Logic Programming (ALP) and Inductive Logic Programming (ILP). These are very active area of research in Artificial Intelligence, Cognitive computing and Knowledge Representation, but there are still many open problems.

We will be covering some of he progress made in abductive reasoning and its application to planning in this course. The Logic-based learning course in the third year focuses instead on the inductive inference and algorithms for learning complex knowledge automatically.



Let's consider a general inference step to be defined in terms of a logical syllogism of the form: *Rule and Case*, given as premises, lead to *Results* as conclusion.

Deduction, has therefore the syllogistic form in which given a rule and some case it is possible to infer conclusions that are already implicitly covered by the premises. The validity of its inference step is guaranteed by the fact that all possible relevant information needed to reach the conclusion is in the premises already. So the principle of a deductive inference is that true premises yield to true conclusions. As it starts from general rules, the conclusions are generated through correct (syntactic) unfolding of the rules given in the premises. In the next few slides we will refer to resolution for clausal logic, and in particular for Horn clauses, as one of the mechanisms for computing this deductive unfolding (or inference step).

Induction can be understood, to a certain extent, as the inverse of deduction. It starts from the "Case" and the observations (taken as conclusions) and infer the general rule that would, consistently with the Case, covers the given observations. So it is the inference of a general rule from the observation of results in given cases. It is called ampliative because it "amplifies the generality of observations into general, learned rules", beyond the boundary of the knowledge expressed in the premises.

Abduction is another kind of inversion of deduction. It starts from the general rule and the observations (taken as conclusions), and infers the cases, called *abductive solutions*, as possible situations in which the given rules would generate the given observations. It is the inference of the cases in which the general rules if applied would lead to the given results.

In his work Pierce emphasized that deduction, induction and abduction are the main three types of valid knowledge inference, and human thinking is the result of combinations of (some of) these types of reasoning. Within the last 15 years researchers have started exploring and investigating possible relationships between these three forms of reasoning and in particular between abduction and induction both in philosophical/theoretical terms as well as in computational terms. This has lead to interesting breakthroughs in the last 10 years in the area of logic-based learning, and new ways of *formalising the notion of scientific inquiries, and supporting machine learning solutions for acquisition of explainable knowledge from data.* This topic is covered in the logic-based learning course taught in the third year.

Example: Electrical Environment

light(l1).
light(l2).
down(s1).
up(s2).
up(s3).
ok(cb1).
ok(outside).
connectedTo(l1, w0).
connectedTo(l2,w4).
live(outside).
connectedTo(w0,w1) ← up(s2).
connectedTo(w0,w2) ← down(s2).
connectedTo(w1,w3) ←up(s1).
connectedTo(w4,w3) ←up(s3).
connectedTo(w3, w5) ← ok(cb1).
connectedTo(w5, outside) ←ok(outside).
lit(L) ←light(L), live(L), ok(L).
live(X) ←connectedTo(X,Y), live(Y).

Deduction

Rule
Case

Results

live(X) ← connectedTo(X,Y), live(Y).
connectedTo(w5, outside) ← ok(outside).
live(outside).
ok(outside).
---
live(w5)

Example: Electrical Environment

```
light(l1).
light(l2).
down(s1).
up(s2).
up(s3).
ok(cb1).
ok(outside).
connectedTo(l1, w0).
connectedTo(l2,w4).
live(outside).
connectedTo(w0,w1) ← up(s2).
connectedTo(w0,w2) ← down(s2).
connectedTo(w1,w3) ←up(s1).
connectedTo(w4,w3) ←up(s3).
connectedTo(w3, w5) ← ok(cb1).
connectedTo(w5, outside) ←ok(outside).
lit(L) ←light(L), live(L), ok(L).
live(X) ←connectedTo(X,Y), live(Y).
```

Case — Inductive

Results

Rules

```
ok(outside).
connected(w5, outside) ← ok(outside).
live(outside).
live(w5).
live(X) ← connectedTo(X,Y), live(Y).
```

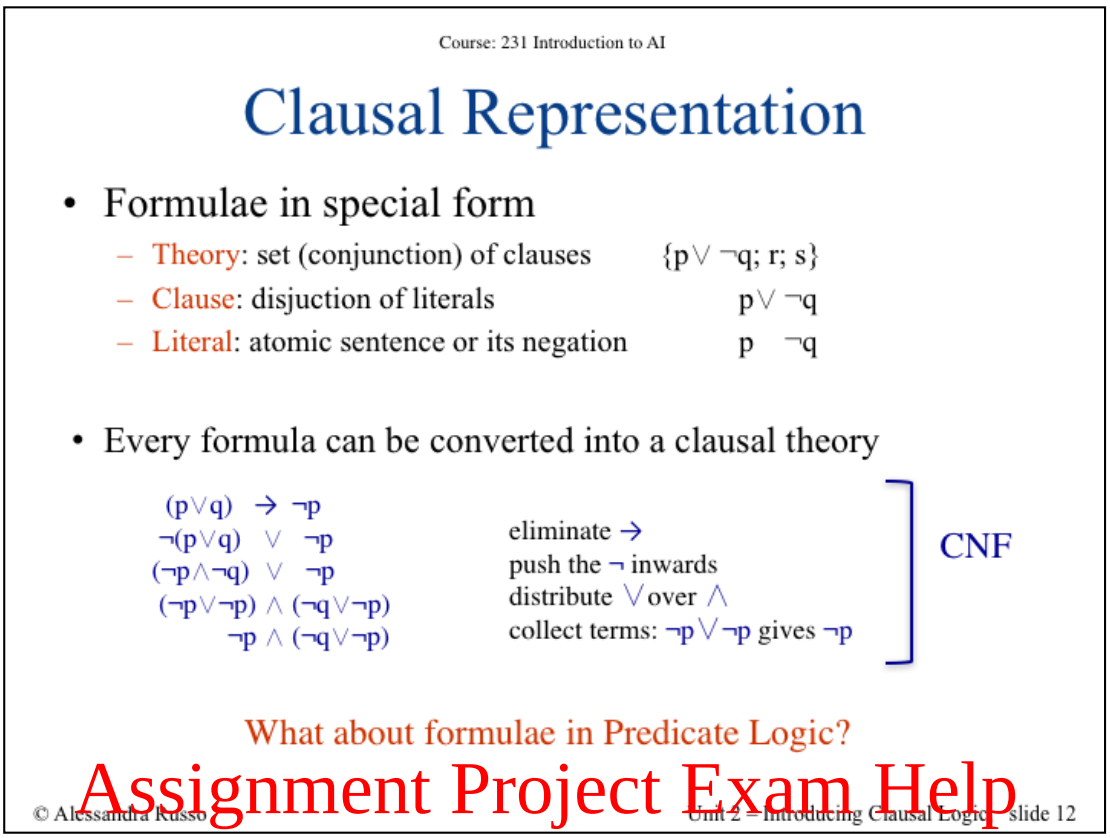
Example: Electrical Environment

```
light(l1).
light(l2).
down(s1).
up(s2).
up(s3).
ok(cb1).
ok(outside).
connectedTo(l1, w0).
connectedTo(l2,w4).
live(outside).
connectedTo(w0,w1) ← up(s2).
connectedTo(w0,w2) ← down(s2).
connectedTo(w1,w3) ←up(s1).
connectedTo(w4,w3) ←up(s3).
connectedTo(w3, w5) ← ok(cb1).
connectedTo(w5, outside) ←ok(outside).
lit(L) ←light(L), live(L), ok(L).
live(X) ←connectedTo(X,Y), live(Y).
```

Abduction

Rule
Results
Case

```
live(X) ← connectedTo(X,Y), live(Y).
connected(w5, outside) ← ok(outside).
live(w5).
ok(outside).
live(outside).
```

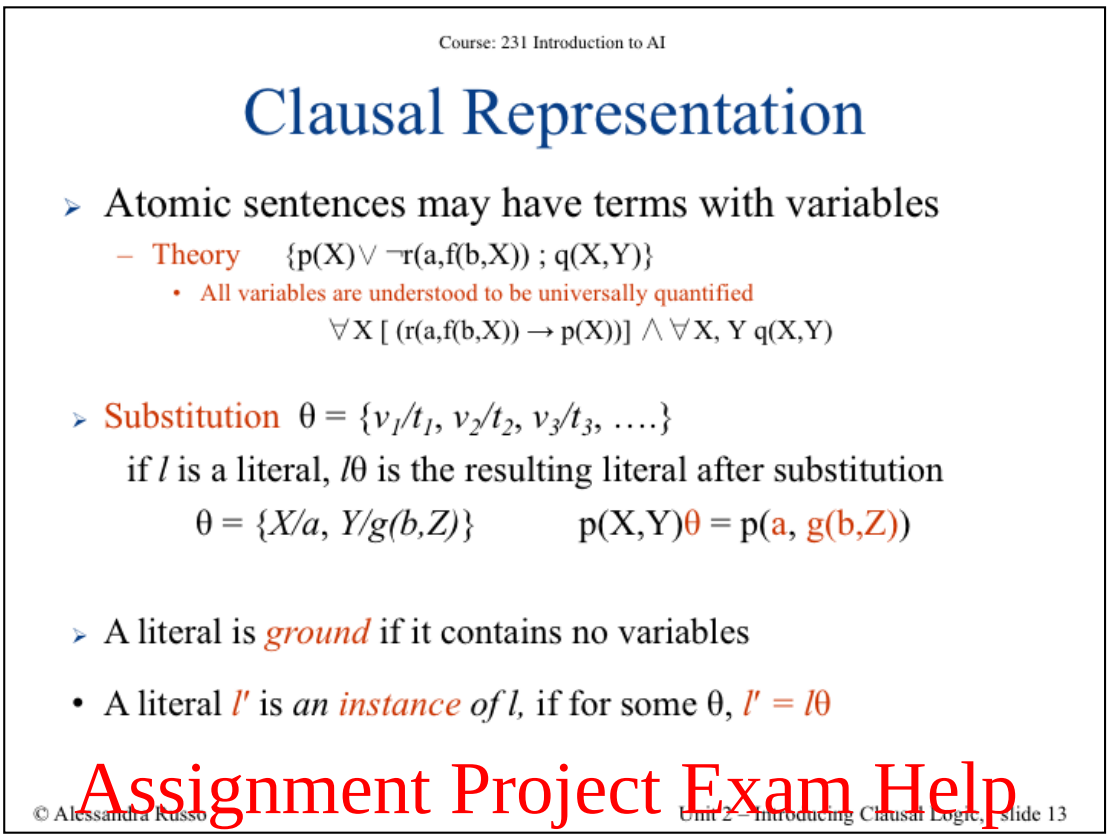


As the focus in this course study the foundation of computational agents as AI solutions to real-world problems, we will restrict ourselves to the subset of predicate logic that is computational tractable and for which efficient automated proof procedures able to compute logical inference exist. The smallest subset of predicate logic, that is computationally tractable, is that of Horn clauses (which is a subset of the Prolog language you have seen at the beginning of this Spring term). We consider also "normal clauses", which extend Horn clauses with the notion of negation as failure, as you have seen in the first part of this course, and we will introduce in the next few lectures Answer Set Programming. The latter is a computational logic language and programming environment that has gained wide attention due of the computational efficiency of the ASP solvers used to find solutions to problems written in this language.

In these next few slides, we recap basic notions of these languages, and related inference mechanisms. We start with summarizing the concepts of *clausal logic* and resolution in both propositional and predicate logic, which are at the basis of Horn clauses and SLD resolution used by Prolog. You have already encountered this notion in your first year Logic course and in the first part of this course. So this should be just a recap for most of you.

A clausal theory is a set of clauses. A clause is a disjunction of literals and a literal is a atomic sentence or a negated atomic sentence, called respectively positive and negative literal. A clausal theory that is finite, can also be seen as a Conjunctive Normal Form (CNF) formula, since all the clauses in the theory are in conjunction with each other and each clause is a disjunction of literals. In the propositional case, it is possible to transform any sentence into an equivalent CNF. An example is given in this slide. So restricting ourselves to consider only formulas in CNF does not semantically restrict the expressivity of the language or of the problems that we can model. The transformation process includes the following steps:

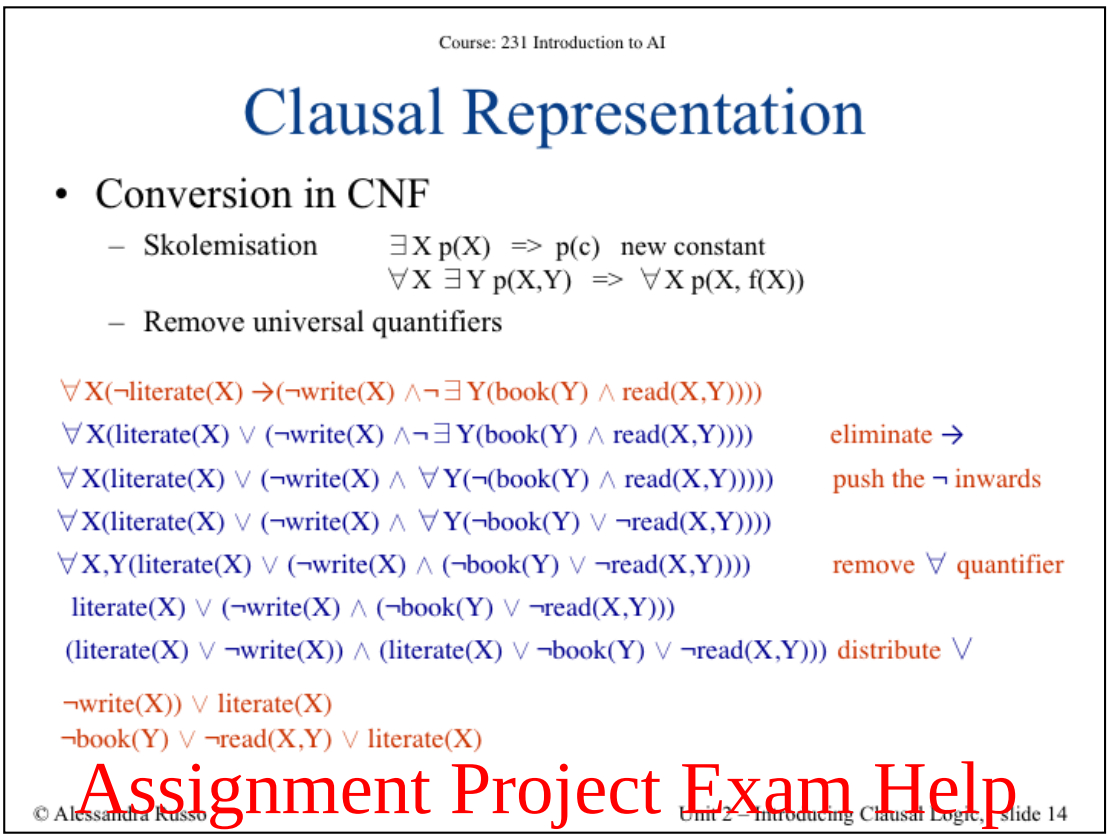
1) change implication into disjunction using the rule: $p \rightarrow q$ is equivalent to $\neg p \vee q$.
2) push negation inwards using the equivalence rules: $\neg(p \vee q) \equiv \neg p \wedge \neg q$ and $\neg(p \wedge q) \equiv \neg p \vee \neg q$.
3) distribute $\vee$over $\wedge$ using the rules: $p \vee (q \wedge s) \equiv (p \vee q) \wedge (p \vee s)$.



The transformation of first-order logic sentences into clausal form is more complex. This is due to the presence of variables and quantifiers. All variables that appear in a first-order logic sentence are quantified either with existential or universal quantifiers. But a first-order clause is a disjunction of literals where all variables that appear in the literals are universally quantified. So the transformation process requires extra steps. In the next slide we describe these steps and we give a couple of examples.

Key concepts, which you may have also seen in previous courses, are the notions of substitution in and instances of first-order sentences. A substitution is a systematic replacement of all occurrences of a term in a sentence by another term. To avoid name clashes variables in the introduced new term should not already appear in the sentence. For example, in the example above, the variable Y cannot be replaced by X or any term that mentions X (e.g. p(a, g(b,X))) would not be a correct substitution.

One of the key important aspects of substitution is the grounding of a clause. This is done by instantiating all the literals that appear in the clause with terms that do not include variables. A ground literal is a literal with no variable (i.e. all its terms are ground terms). So a ground clause generated from a given non-ground clause by replacing all terms in the latter with ground terms is called instance of the (non-ground) clause.

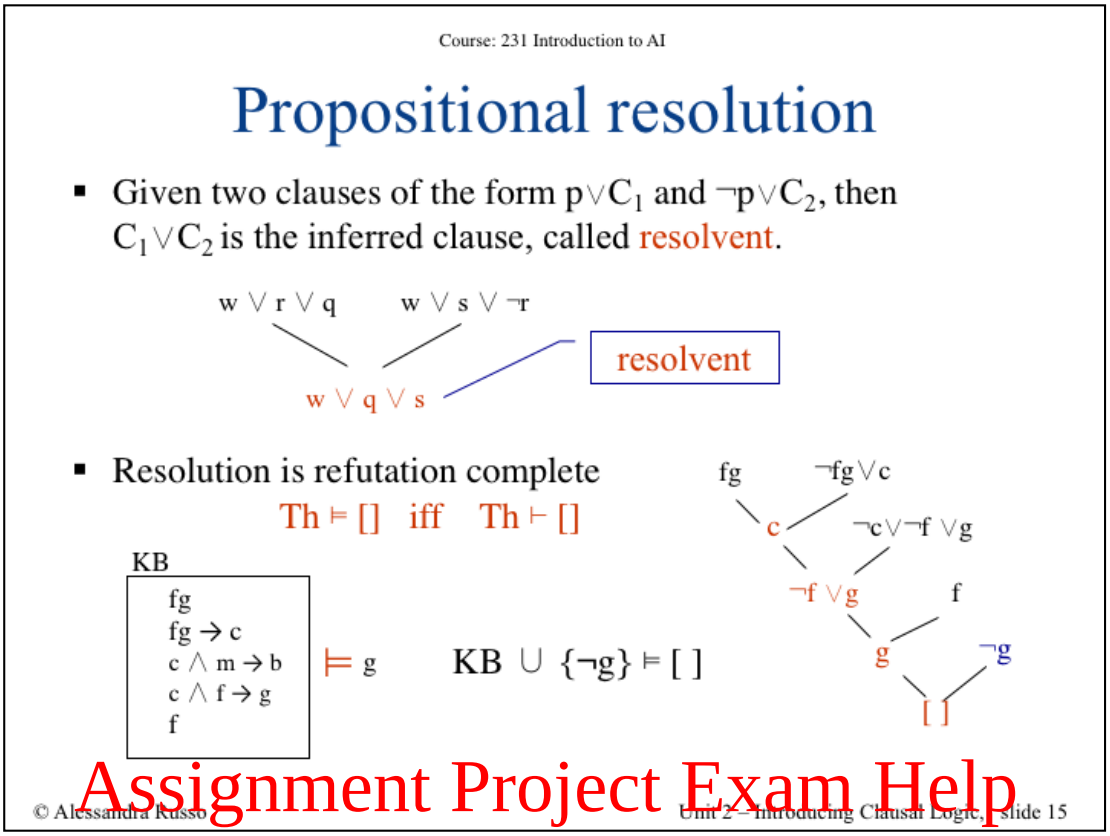


Two key steps are needed when transforming a first-order sentence into a clausal theory or CNF formula. As in the propositional case, implication should be changed into disjunction. Negation can then be pushed inwards using also the additional rules: $\neg \exists X p(X) \equiv \forall X \neg p(X)$ and $\neg \forall X\, p(X) \equiv \exists X \neg p(X)$. This step helps eliminating existing existential quantifiers but, at the same time, may introduce also new existential quantifiers. Remember that the target clausal form assumes all variables to be universally quantified. So, as second step, all remaining existentially quantified variables can be eliminated by introducing Skolem term. When a subformula $\exists Y\, F[Y]$ exists in a given formula (note that here *F* can represent more than just a single predicate and the *Y* in square brackets denotes that the variable *Y* appears in the formula *F*), then the existential quantifier can be removed by introducing a Skolem term in the following way: i) if the subformula is not in the scope of any universal quantifier then use a new skolem constant name and substitute all occurrences of Y with this new Skolem constant name, ii) if the subformula occurs within the scope of one of more universal quantifiers for variables $X_1, X_2, .. X_n$, then introduce a Skolem term $sk(X_1, X_2, .. X_n)$ and substitute all occurrences of Y with this new term.

(i.e. Note that the new Skolem constants and Skolem terms must not appear already anywhere else in the theory and are essentially assumed to extend the language (or signature) of the theory).

Once all existential quantifiers have been removed, all universal quantifiers can then be moved to the front of the sentence using the property that $(\forall X\, \varphi_1(X)) \wedge \varphi_2$ is equivalent to $\forall X(\varphi_1(X) \wedge \varphi_2)$ after appropriate variables renaming in case X already appears in φ_2.

Finally, $\vee$ can be distributed over $\wedge$. An example is given in this slide.



Once sentences are transformed into set of clauses, specific inference algorithms can be used. One of the most well known is ***resolution***. A resolution step is applied between two clauses that include respectively literals opposite in sign (i.e. a positive literal and its corresponding negative literal). The inference of a resolution step is a new clause given by the disjunction of all the literals that appear in the two given clauses with the exception of the literals opposite in sign. This new clause is called the ***resolvent clause***.

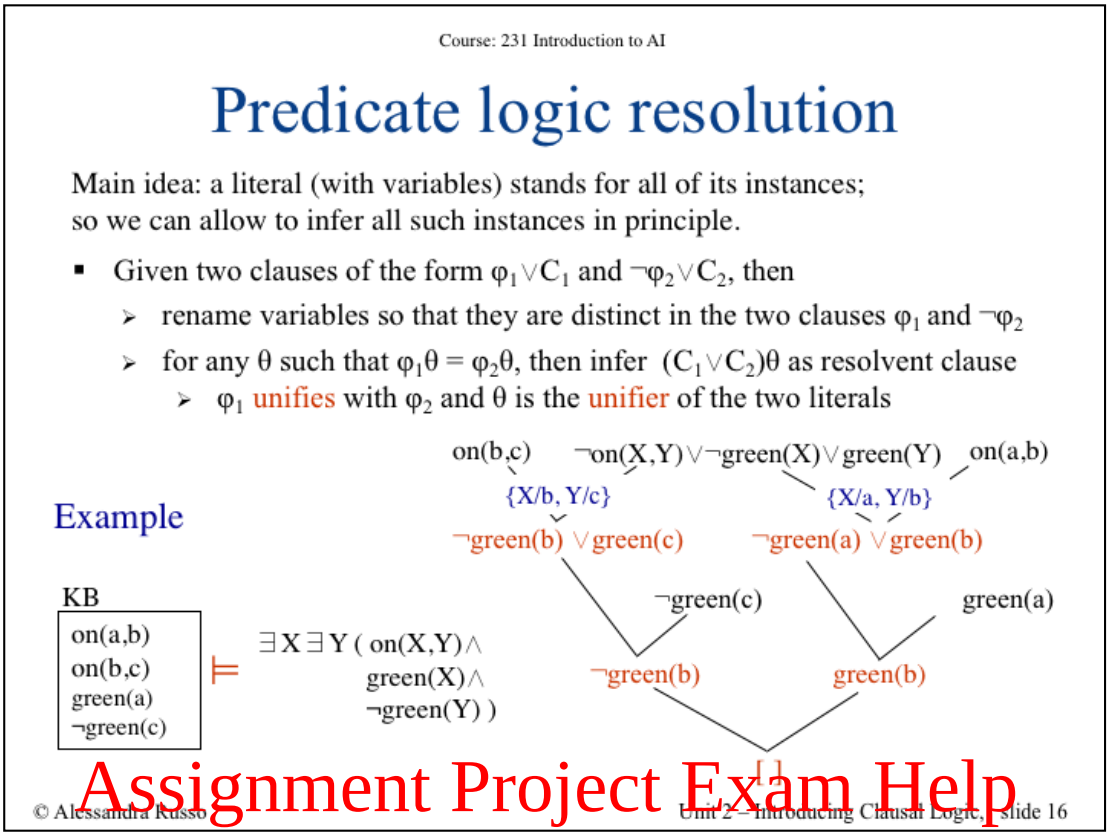
We have mentioned in previous slides that inference is sequential application of inference rules that are applied to given premises and/or conclusions generated in previous steps in the sequence. A derivation of a clause c from a given clausal theory Th is therefore a sequence of clauses $c_1,\ldots c_n$ where c_n is the clause c and every other clause c_i is either in Th or is the resolvent of two earlier clauses in the sequence. Resolution is a *sound* proof procedure, namely

$$Th \vdash c \text{ by resolution} \qquad \text{THEN} \qquad Th \models c$$

The opposite does not hold directly, but it holds in a refutation way. For instance, consider the case of $p \models p \vee q$. Resolution cannot be directly applied to the given clausal theory Th = {p} and infer $p \vee q$. However if the problem is transformed into a refutation problem (or proof by contradiction), the resolution proof procedure can be applied. **So resolution is both sound and refutation complete**.

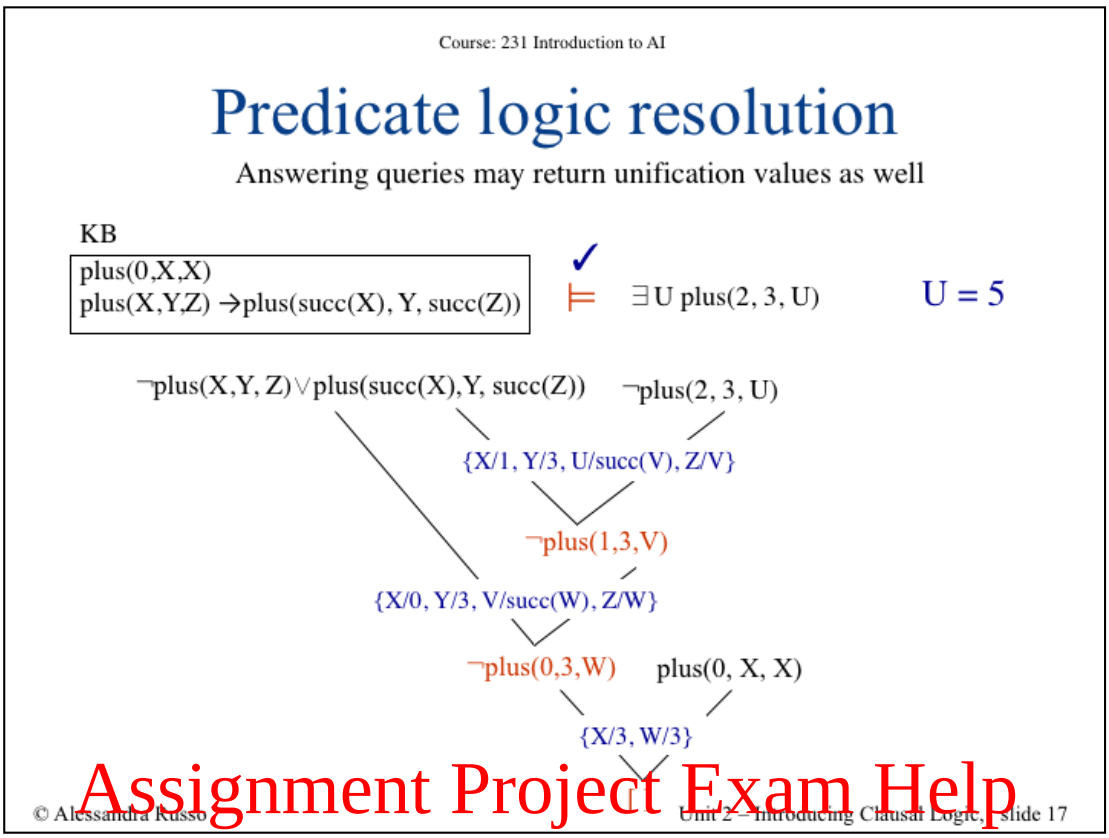
For example, the inference task $p \vdash p \vee q$ can be expressed as a refutation problem $\{p , \neg(p \vee q)\} \vdash [\,]$, where [ ] (empty clause) denotes *false*. Transforming it into clausal form, we get the set of clauses {p, ¬p, ¬q}. We need to show that $\{p, \neg p, \neg q\} \vdash [\,]$. Note that as shown also in the above examples not all clauses need to be used in the derivation. So, it is easy to see that this is indeed the case. So to use resolution, a logical inference task has to be "rephrased" as a refutation task. This is the basic principle underlying SLD refutation in Prolog, where a given query is in essence a negated goal. More to come later on when we recap the notion of SLD.

In the above slide an example of simple propositional resolution is given. Note that the given query (i.e. g) or clause that we want to prove is first negated and added to the existing set of clauses and the derivation tries to derive the empty clause.



Resolution in predicate logic is more complex. It is still based on the idea of "resolving" opposite literals that appear in two clauses but variables play an important role here. Literals may have clearly unground terms. In this case they are understood as standing for all possible instances so in principle the resolution could happen by referring to any of such instances. Identifying which instance to use is the role of the unification step. Firstly, all variables occurring in the two clauses to resolve should be renamed with unused variables to avoid name clashes. Then considering two opposite literals in the two clauses, if it is possible to find a substitution that, applied to both these two literals, makes them equal, then the resolution step can be applied and the substitution has to be systematically applied to all the occurrences of the variables in the literals left in the resolvent. For instance, in the example above, the left most literal on(b,c) can resolve with the opposite literal ¬on(X,Y) in the top middle clause by means of the substitution {X/b, and Y/c}. These two literals are then said to "unify" under this substitution and the substitution is applied to the rest of the literals in the resolvent clause. This explains the resolvent ¬green(b) ∨ green(c). And so on.

Note that in this example of resolution the given query has been negated, transformed into a clause and added to the given knowledge base (KB) expressed as set of clauses. The negation had the effect of eliminating the existential quantifiers, hence no skolemisation has been applied here.



The previous example has shown that the resolution mechanism can allow us to answer existential types of questions. But what about if we actually want to find out particular values for which our query is derivable from the given set of clauses? The resolution process uses unification and this same unification constructs during the proof a substitution function that defines, by the end of the inference, the values of the variables in the given query.

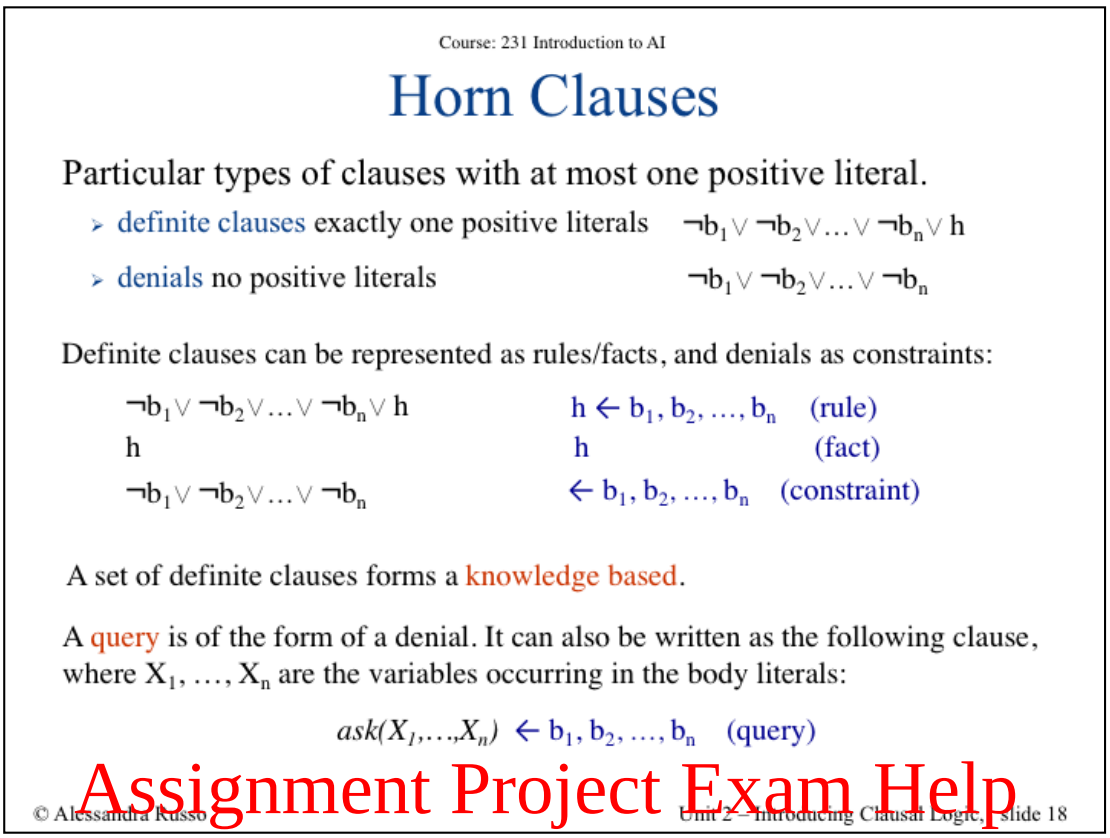
Consider the above example. In this case, the query is negated and transformed into a clause with variable U universally quantified. The blue labels in the derivation show the unification applied to each resolution step. Since U = succ(V), and V= succ(W) and W = 3, the three unifications give

U = succ(succ(3)) which is indeed 5.

Similar principles happen in SLD resolution in Prolog when queries with variables are posed to a Prolog program and the Prolog returns yes with the value of the variables that appear in the query.

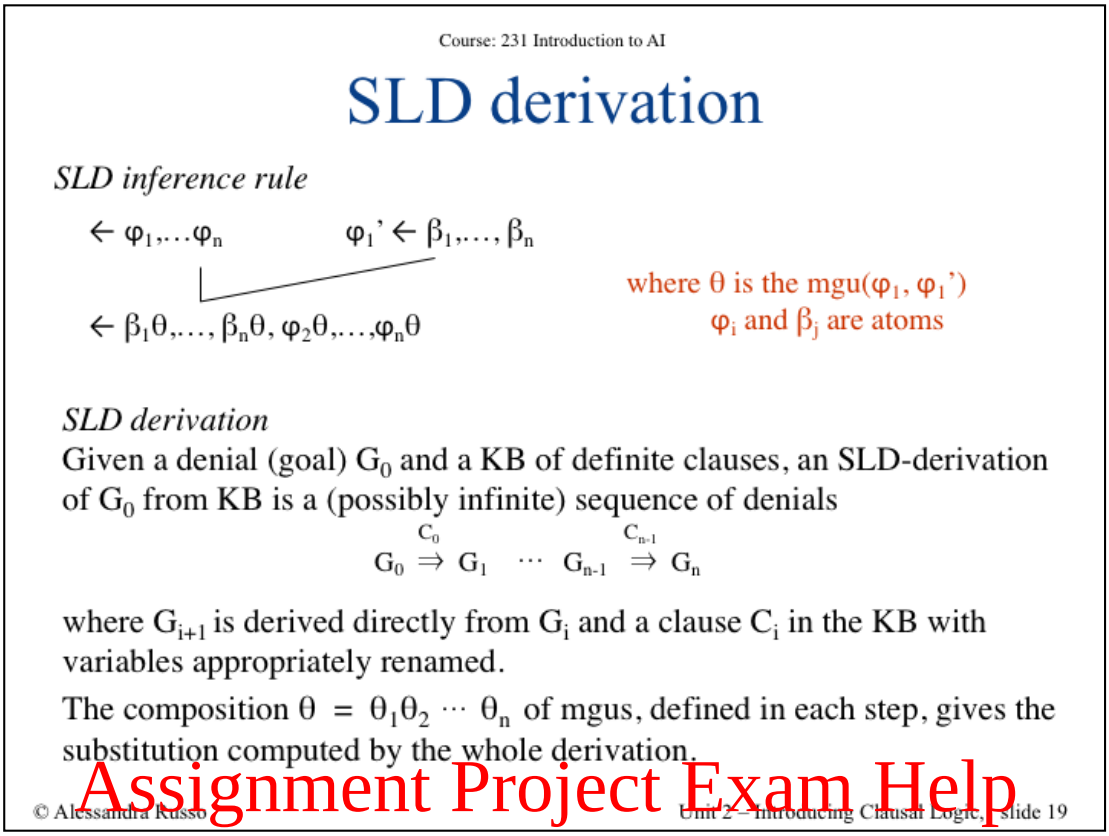
However, it is possible to construct resolutions that never terminate. This is often the case when function symbols appear in the given clauses and the resolution process is applied in a kind of "depth-first" manner.

To tackle this problem a possible way for making resolution more efficient is to use the notion of Most General Unifier (MGU). This is the most general unifier that can be constructed between two given clauses so that any other unifier between these clauses can be generated by concatenating another unifier function to the MGU. This notion goes beyond the content of this course.



We have seen so far how clausal representation is supported by inference mechanisms that are refutation complete. The notion of clausal theory and resolution is at the heart of declarative programming. Declarative programs are written using a special types of clauses, called *definite clauses* and queries posed to such a program are assumed to be *denials*. A definite clause is a clause with exactly one positive literal. Denials are instead clauses with zero positive literals. Denials are also referred to as integrity constraints. As we will see in the next lecture, denials play an important role in the abductive inference.

A definite clause can be read as a *rule* of the form shown in the slide. The positive literal is the head of the rule and the atoms of the negative literals form the body of the rule. Definite clauses composed of just the positive literal are also called *facts* (rules with empty body). Integrity constraints can also be understood as special rules with *false* as head of the rule. We will use the notation shown in blue from now to represent definite clauses and denials.

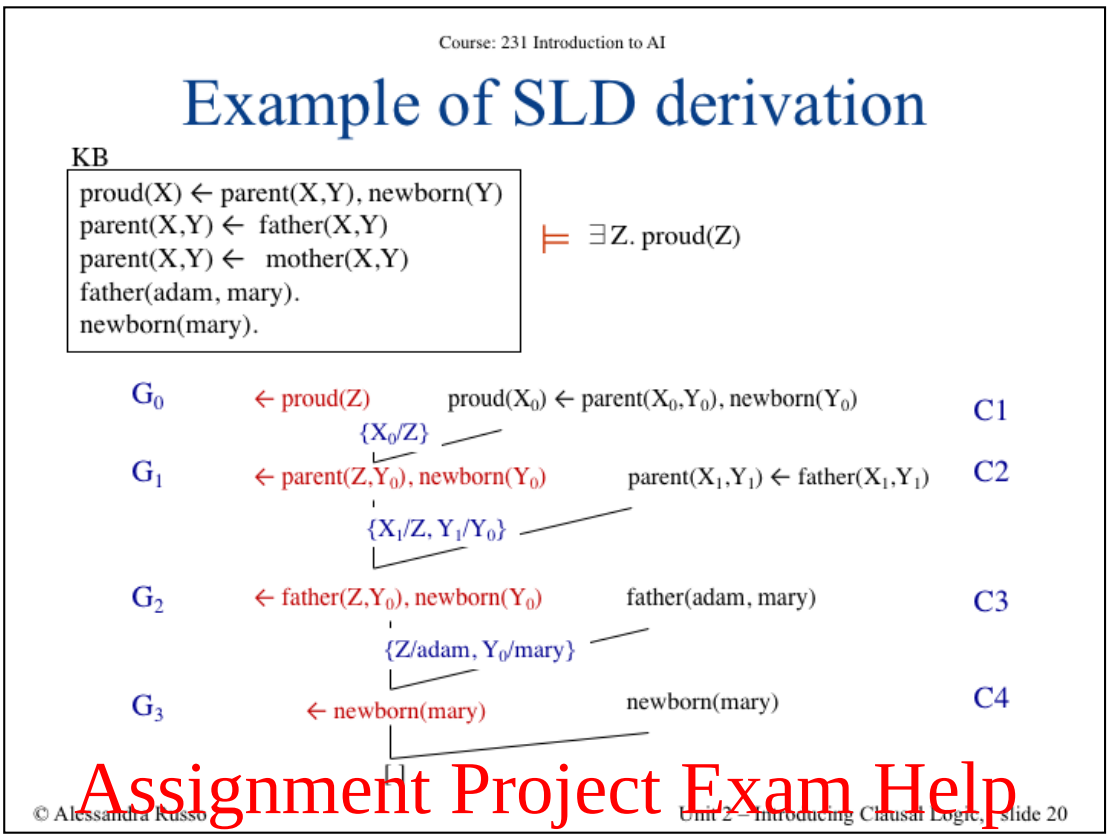


SLD inference procedure, used by Prolog, is a specialized version of clausal resolution since in the previous slides, for classical logic. It exploits the fact that the given knowledge base (KB) is a set of definite clauses and not general clauses. The SLD inference rule is always between a denial and a definite clause. It selects the left most literal in the denial, unifies it with the head of a clause and returns a new denial where the selected literal is now replaced by the body conditions of the clause.

At the beginning, the query is the first given denial $\leftarrow b_1, b_2, \ldots, b_n$ (remember SLD is still a resolution proof method and as such is still refutation based). Queries are also sometime formalised as $ask(X_1,\ldots X_n) \leftarrow b_1, b_2, \ldots, b_n$, where $X_1,\ldots X_n$ are the variables that appear in the body of the denial. So, given a KB, expressed as a set of definite clauses, and given a query (the first denial), the answer to the query is computed by constructing an SLD derivation.

An SLD derivation is a linear sequence of applications of SLD inference rule that generates a sequence of denials. The first denial is the given query. Any subsequent denial is generated by the application of the SLD inference rule between the previously generated denial and a clause in the KB. This process continuous until an empty [ ] denial is generated. This is formally defined in the slide above.

Note that, more than one atom in the generated denial may unify with literals in the chosen clause, so by convention SLD inference rule uses a selection fnction that picks always the left most atom in a denial before applying the rule. The name SLD stays indeed for *Linear Resolution for Definite clauses with Selection function.*



The above is a simple refutation example. Note that the background program describes only "positive knowledge" — it does not state who is not proud. Nor does it convey what it means for someone not to be a parent. The problem of expressing negative knowledge will be investigated in subsequent lectures of this course. As part of this lecture we will briefly show how SLD resolution copes with set of rules that include negation as failure. Later in the course you will learn about semantics of theories with negation as failure.

A sequence of G_0, G_1,…,G_{n+1} goals that terminates with an empty clause is a successful SLD derivation. Different Horn clauses can be chosen during the derivation. Different choices would lead to different SLD derivations. Some of these will also not be successful derivations. An SLD derivation fails when the selected sub-goal cannot be unified with any clause and therefore the sequence cannot be extended further.

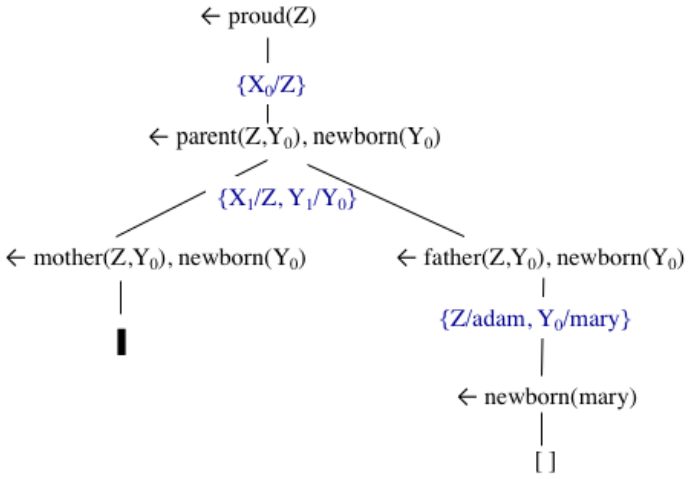
So, by definition, an SLD derivation of a goal G_0 is a failed derivation if the last element in the derivation is not the empty clause and it cannot be resolved with any other clause given in the initial theory (KB).

An example of failed derivation could be constructed for the same goal and KB given in this slide, if the third clause in the KB is chosen as clause C_2 instead. In this case the next subgoal G_2 would be the denial ← mother(Z,Y_0), newborn(Y_0). The selected literal mother(Z,Y_0) would in this case not be unifiable with any other clause in the KB.

Course: 231 Introduction to AI

SLD Trees

A denial can unify with more than one clause. So multiple SLD derivations could be computed:

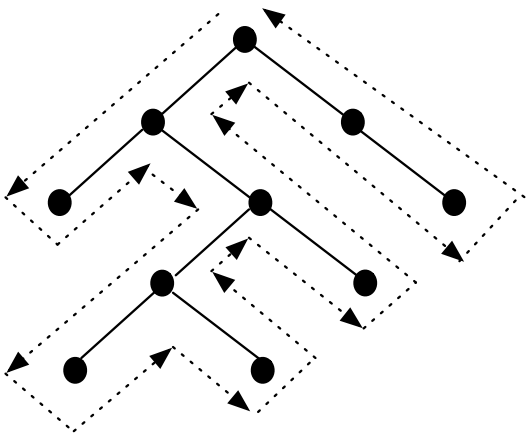


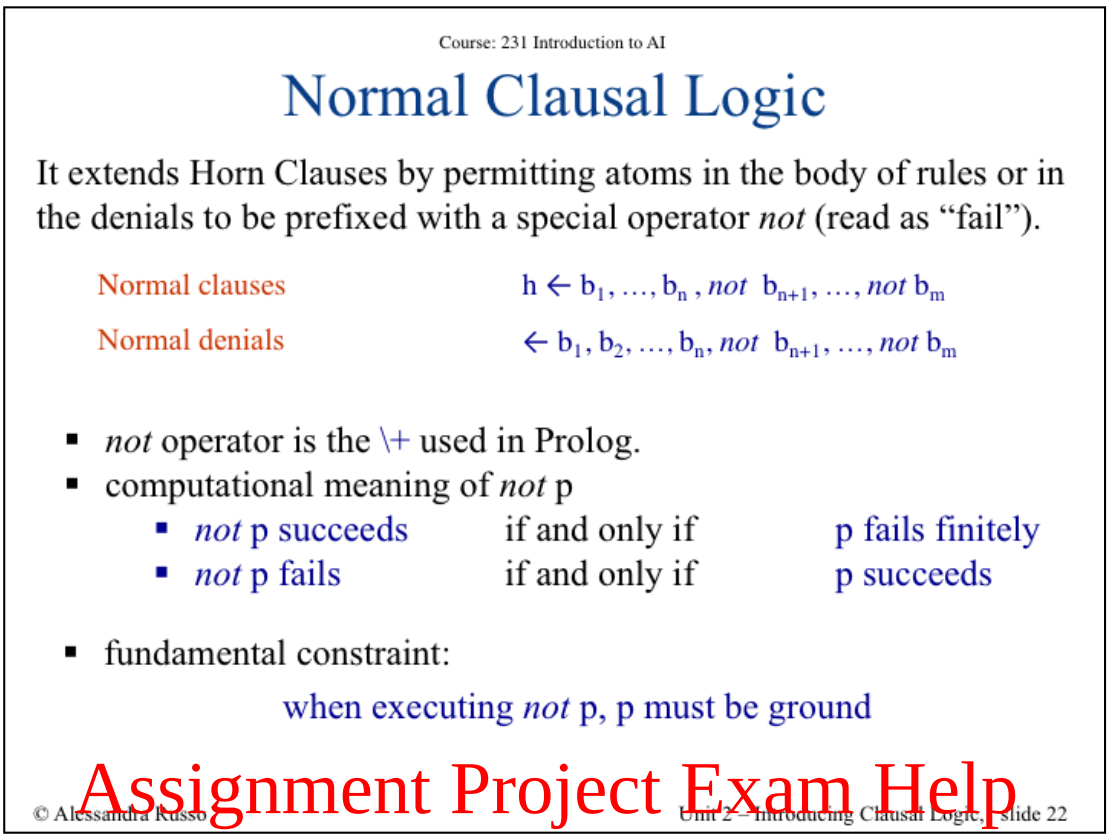
If the (inferred) denial can be resolved with more than one clause in the KB, then different SLD derivations can be constructed. All such derivations together form a (possibly infinite) SLD tree of a given denial G_0. An example of SLD tree is given in the slide. Note that only the denials generated by the application of SLD rules are shown in this SDL.

An SLD tree essentially represents the search space for SLD derivations. Each branch of the tree is an SLD derivation. Generating one or all (where possible) SLD derivations means performing a systematic search over the search tree of such inference process. Prolog exploits the ordering in which the clauses are written in the given program (i.e. KB), picking first the clauses that appear at the top of the program. The selection strategy is from top to bottom. This imposes the ordering on the edges descending from a node of the SLD-tree. Following this selection criteria, the tree is explored in a **depth-first manner**. For a finite SLD-tree this strategy is complete. Whenever a leaf node of the SLD-tree is reached the traversal continues by backtracking to the last preceding node of the path with unexplored branches (see figure below). If the empty clause is reached the answer substitution of the completed SLD derivation is returned.

Notice, however, that an SLD tree may be infinite. This is why sometimes Prolog computation do not terminate.

Can you think of an example where the tree is infinite (e.g. it has an infinite branch)?





In the first part of this course you have also seen the notion of normal clauses and the semantics of KB written as set of normal clauses. The next few slides summarise this concept again, in particular from the point of view of its related inference process, called SLDNF, with the purpose of introducing the notion that we will be using.

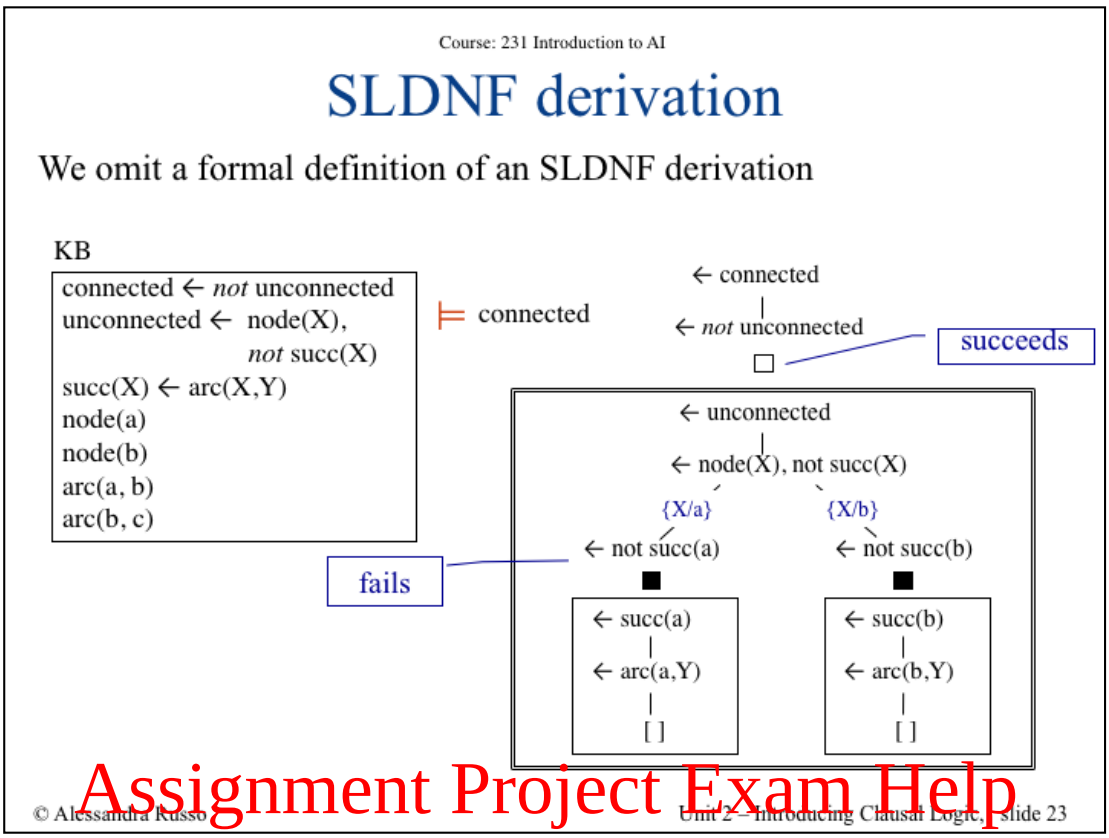
Normal clauses include literals that have a special operator, denoted as "\+", and called "fail" operator or *negation by failure*.

A literal "*not* p" in a normal clause can in fact be read as "fail to prove p". The name "negation by failure" is related to the semantics of this operator, which in turns is related to the notion of Clark Completion semantics. In very simple terms, "*not* p" means semantically "state **not p** to be true in the model of a given program because the program has failed to prove p". This is intuitively related to the notion of Least Herbrand Model and the property that everything that is provable from a program is true in the model of the program. So failing to prove p means stating that p is false and therefore that the negation of p is true (but omitted) in the model of the program.

Atoms prefixed with the negation by failure operator can also appear in denials. We refer to these as normal denials. But you can immediately notice that a denial with index n=0 and index m=1 is of the form ← not p(X) (as the atoms in the clauses may be predicates). **This type of expression cannot be evaluated if the atom p(X) is not ground. When the inference process generates a denial of this form, the variable X should have already be unified in the previous steps of the derivation. In other words, when evaluating ← not p, the atom p must be ground.**

If this is not the case then we say that the derivation "floundered", meaning that it does not know the answer. In the next slide we show an example that illustrates this.

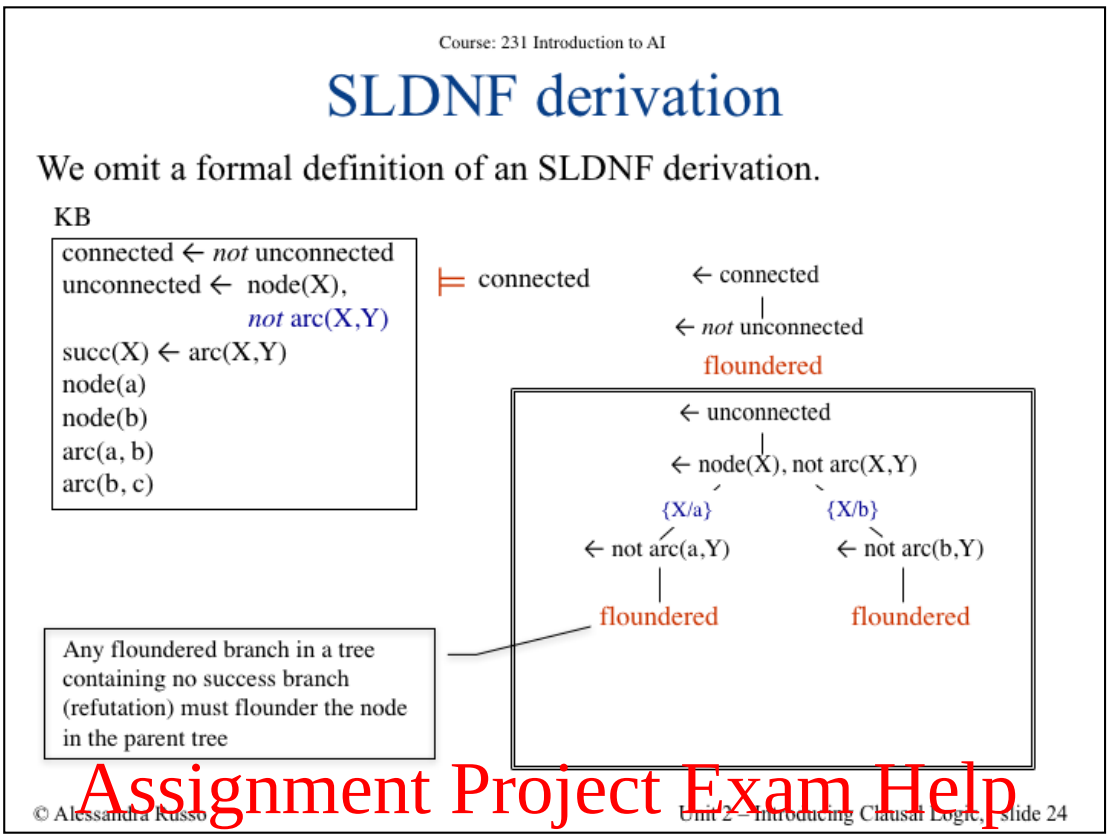
Note that in normal clausal logic the fail operator never appears in the head of a rule, as this would correspond asking to prove what should not be proved, whereas clausal theories define what should be provable!



In the above example derivation, to evaluate the given query (denial), the second denial ← *not unconnected* is generated by resolving it with the first cause in the KB. Now to prove this negated literal, a new derivation is created. This is indicated with the double line box. The double line box derivations are failure derivations. This means that all possible ways of deriving the denial ← *unconnected* must fail.

Given the KB, there is only one rule that unifies with ← *unconnected*. So at least one of its body literals will have to fail. The denial ← node(X), not succ(X) is generated, then two possible derivations can be constructed for the literal ← node(X), as there are two possible unifications with the two facts node(a) and node(b) in the KB. In each of these two possible derivation branches a new denial with negated literal is generated. Consider the left branch. The denial ← not succ(a) triggers a new failure derivation. This means that ← succ(a) has to fail (i.e. the derivation must not finish with an empty close) for ← not succ(a) to succeed. But, in this case the derivation does finish with an empty clause, so ← succ(a) succeeds and therefore ← not succ(a) fails. Similarly, in the right branch.

Now, because ← not succ(X) fails the derivation of the denial ← *unconnected* fails too, so the derivation of ← *not unconnected* succeeds, and so does the initial goal denial ← *connected*.



In the above example a slight modification has been made in the second clause of the KB. Note that the condition "**not arc(X,Y)**" mentions a variable Y that does neither appear in the head nor in a previous positive literal in the body of the rule. In this case the SLD derivation will eventually generate the denial goal ← not arc(a,Y) where Y is not grounded and the derivation will flounder.

To prove such a denial goal the process should try to fail to prove ← arc(a,Y) for any value of Y. Because in principle there are infinitely many possible values of Y, this could, in principle, require searching through possibly infinitely many derivations in order to show that does not exist a value for Y for which arch(a,Y) succeeds. This is why the SLDNF requires that any negated denials ← not φ will need to be grounded. If φ includes at least an unground variable, the derivation of such denial will flounder.

We will see that in the context of Answer Set Programming, problems of floundering and computational loops that goal-directed Prolog proof procedure suffer of, are no longer an issue. Floundering cases are controlled but the requirement that rules in ASP programs are *safe*. We will see this concept later on in the course.

Hopefully, in the first part of this case you have seen the semantics of KB written as set of definite clauses (i.e. Least Herbrand Model semantics) or as set of normal clauses (i.e. Stable Model semantics). The question is why bother with negation as failure? The answer is that it provides a more flexible means for representing knowledge. In the 4th year course of Knowledge Representation you will hopefully see in more detail how negation by failure is an ideal operator for expressing default assumptions and capture default reasoning. We will see later in the course, that it is indeed useful for expressing notion of persistency of effect of actions in the real world, which is key when we address a planning problem.

Summary

- Propositional and predicate logic.
- Types of formal reasoning:
  deduction, abduction and induction
- Resolution: one of the main deductive proof procedures used in computational logic.
- Recap of Horn clauses and SLD resolution.
- Illustration of SLDNF for normal clauses