Languages and Algorithms for Artificial Intelligence

Module 1

Matteo Donati

August 13, 2021

Contents

Chapter 1

Introduction

1.1 Artificial Intelligence

There exist many definition of artificial intelligence. Some of these are the following:

Thinking Humanly	Thinking Rationally
"The exciting new effort to make computers think ... *machines with minds*, in the full and literal sense." (Haugeland, 1985)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985)
"[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998)
"The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"AI ...is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

There exist two main approaches to define an artificial intelligence system:

- A **symbolic approach**, based on logics, which can provide explanations for decision that have been made.
- A **connessionist approach**, based on machine learning, which is suitable to solve problems that, indeed, can be solved by specific algorithms.

1.2 Logic

Logic is the systematic study of the form of valid inference, and the most general laws of truth. A valid inference is one where there is a specific relation of logical support between the assumptions of the inference and its conclusion. In particular, there exist different types of logic:

- **Classical logic**, which considers truth and inference (e.g. if X and Y are true, can I deduce that also W is true?). Normally, it is interested in deriving new facts from known ones.
- **Intuitionistic logic**, which considers constructive proofs (e.g. if I can construct a proof for X and Y, can I construct a proof also for W?).
- **Linear logic**, which considers resources (e.g. if I have X and Y, can I exchange them for W?).
- **Epistemic logic**, which considers knowledge and belief (e.g. if I believe that you know X and Y, can I conclude that I believe W?).
- **Temporal logic**, which considers the evolution in time (e.g. is X and Y will happen, can I conclude that W could happen?).

Computer science derives from logic, and in particular:

- Logic is related to the foundation of computation and of declarative programming languages.
- Logic is related to the foundation of theoretical computer science (computability).
- Logic allows proving of software's correctness.
- Logic can be considered as a paradox free language (in contrast with natual language).

Moreover, artificial intelligence benefits from logic. In particular:

- Logic represents a tool for expressing human reasoning.
- Logic is related to the foundation of a classic approach to artificial intelligence and of declarative languages used in artificial intelligence.

1.3 Declarative Programming

Bob Kowalski gave the following definition:

$$Algorithm = Logic + Control$$

In a traditional programming language one has to specify both the logic part and the control part. However, in declarative programming, one has to specify only the logic part since the control is taken care of by the machine. In particular, declarative programming includes logic programming, constraint programming and functional programming.

Chapter 2

Truth

Truth is most often used to mean being in accord with fact or reality, or fidelity to an original or standard. Truth is also sometimes defined in modern contexts as an idea of authenticity. In the real world, truth is associated to the sensible world. In particular, repeatable experiment defines a truth and the truth changes as the considered world changes.

2.1 Logical Consequence

Given a set of sentences (hypothesis or premises), $\Gamma = F_1, F_2, \ldots, F_n$, and a conclusion, F, then F is a logical consequence of Γ ($\Gamma \models F$), if for different interpretations of the formulas in the various possible worlds, it is always true that if all the formulas in Γ are true (in the world under examination) then also F is true.
Therefore, truth is always defined with respect to a specific world and a given interpretation of symbols in that world. On the other hand, logical consequences do not change when the world changes.

2.2 Logical Equivalence

Two formulas, F and G are said to be logically equivalent if and only if F is a logical consequence of G and G is a logical consequence of F.
Therefore, truth is always defined with respect to a specific world and a given interpretation of symbols in that world. On the other hand, logical equivalences do not change when the world changes.

Chapter 3

Propositional Logic

In general, for any formal system (programming language, logic, etc.) there exist a syntax and a semantics. In particular, the former allows the definition of the rules which tell how the well formed sentences are constructed, while the latter is used to understand the meaning of the well formed sentences.
Propositional logic is the simplest logic and concerns propositions, i.e. statements about reality which can be true or false. Examples of propositions are the following:

P_1: the snow is white
P_2: it's raining today
P_3: this course is boring

Each P_i can be either $true$ or $false$ and the values associated to each proposition is called *truth value*. These propositions can be also connected together using connectives, thus, the language of propositional logic has an alphabet consisting of (i) propositional symbols, p_0, p_1, ..., p_n, (ii) connectives, $\wedge$, $\vee$, $\rightarrow$, $\neg$, $\leftrightarrow$, $\bot$ and (iii) auxiliary symbols: (,). The propositional symbols and $\bot$ are called **atoms**.
if-then if and only if

3.1 Syntax

The syntax of propositional logic can be defined in different ways. The first one is a recursive definition of well-formed formulas:

- An atom is a formula.
- If S is a formula, $\neg S$ is a formula (negation).
- If S_1 and S_2 are formulas, $S_1 \wedge S_2$ is a formula (conjunction).
- If S_1 and S_2 are formulas, $S_1 \vee S_2$ is a formula (disjunction).
- All well-formed formulas are generated by applying the above-mentioned rules.

Another definition uses the **Backus-Naur Form** (BNF):

$$\begin{array}{rcl} \langle \text{formula} \rangle & ::= & \text{atomic proposition} \\ & | & \neg\langle \text{formula} \rangle \\ & | & \langle \text{formula} \rangle \wedge \langle \text{formula} \rangle \\ & | & \langle \text{formula} \rangle \vee \langle \text{formula} \rangle \\ & | & \langle \text{formula} \rangle \rightarrow \langle \text{formula} \rangle \\ & | & \langle \text{formula} \rangle \leftrightarrow \langle \text{formula} \rangle \\ & | & (\langle \text{formula} \rangle) \end{array}$$

Finally, a third inductive definition, and probably the most important one, is the following:

$$\begin{array}{l} p_i \in X (i \in N), \bot \in X \\ \varphi, \psi \in X \Rightarrow (\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi) \in X \\ \varphi \in X \Rightarrow (\neg\varphi) \in X \end{array}$$

In particular, the set $PROP$ of propositions is the smallest set X with the above-mentioned properties.

3.2 Semantics

Semantics in propositional logic is divided into *model theory* (what is true) which uses truth tables and *proof theory* which is based on natural deduction (what is provable).

3.2.1 Model Theory

In order to define the semantics of propositional logic formulas it is necessary to define the meaning of the propositional logic symbols. In particular:

- Given a propositional logic formula G, composed by the atoms $A_1, \ldots, A_n$, an interpretation I of G is an assignment of truth values to $A_1, \ldots, A_n$. In particular:
  - The value associated with each atom, A_i, can be either *true* or $false$ but never both.
  - Given an interpretation I, G is said to be *true* in I if and only if G is evaluated to *true* in the interpretation.
  - If G has n distinct atoms, then it is possible to define 2^n distinct interpretations for the atoms in G.
  - Given an interpretation I, if G is *true* under I, then I is a model of G ($I \models G$).

  Moreover, the semantics of the connectives is defined by using *truth tables*:

  logical consequence

P_1	P_2	$\neg P_1$	$P_1 \wedge P_2$	$P_1 \vee P_2$	$P_1 \rightarrow P_2$	$P_1 \leftrightarrow P_2$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

* • If there exists at least an interpretation I such that $I \models F$, then F is satisfiable.

- Given a formula, G, this is said to be **valid** if and only if it is *true* in all its interpretations ($\models G$). In this case, G is a **tautology**.
- Given a formula, G, this is said to be **inconsistent** if and only if it is $false$ in all its interpretations. In this case, G is said to be **unsatisfiable**. *
- Propositional logic is **decidable** by nature; there exist a terminating method to decide whether a formula is valid or not. In order to do so, one can enumerate all possible interpretations (2^n) and for each of them evaluate the formula.
- Two formulas, F and G are said to be **logically equivalent**, $F \equiv G$, if and only if the truth values of F and G are the same under every interpretation of F and G. Some useful equivalences are the following:

logical consequence in both directions

$$
\begin{array}{rl}
(P \wedge Q) \equiv (Q \wedge P) & \text{(commutativity of } \wedge\text{)} \\
(P \vee Q) \equiv (Q \vee P) & \text{(commutativity of } \vee\text{)} \\
((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R)) & \text{(associativity of } \wedge\text{)} \\
((P \vee Q) \vee R) \equiv (P \vee (Q \vee R)) & \text{(associativity of } \vee\text{)} \\
\neg(\neg P) \equiv P & \text{(double-negation elimination)} \\
(P \rightarrow Q) \equiv (\neg Q \rightarrow \neg P) & \text{(contraposition)} \\
(P \rightarrow Q) \equiv (\neg P \vee Q) & \text{(implication elimination)} \\
(P \leftrightarrow Q) \equiv ((P \rightarrow Q) \wedge (Q \rightarrow P)) & \text{(biconditional elimination)} \\
\neg(P \wedge Q) \equiv (\neg P \vee \neg Q) & \text{(De Morgan)} \\
\neg(P \vee Q) \equiv (\neg P \wedge \neg Q) & \text{(De Morgan)} \\
(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R)) & \text{(distributivity of } \wedge \text{ over } \vee\text{)} \\
(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R)) & \text{(distributivity of } \vee \text{ over } \wedge\text{)}
\end{array}
$$

- Given a set of formulas, $\{F_1, \ldots, F_n\}$, and formula G, G is said to be a **logical consequence** of $F_1, \ldots, F_n$ ($(F_1 \wedge \cdots \wedge F_n) \models G$) if and only if for any interpretation I in which $F_1 \wedge \cdots \wedge F_n$ is *true* then also G is *true*. In particular, $F_1, \ldots, F_n$ are called **axioms** or **premises** for G.
- **Normal forms**: standard ways of writing formulas. There exist two main types of normal forms:

also called clausal normal form

  - **Conjunctive normal form** (CNF). A formula F is in CNF if and only if it is in NNF and it has the form $F \triangleq F_1 \wedge F_2 \wedge \cdots \wedge F_n$, where each F_i is a disjunction of literals. If F is in CNF, then each F_i is called **clause**.
  - **Disjunctive normal form** (DNF). A formula F is in DNF if and only if it is in NNF and it has the form $F \triangleq F_1 \vee F_2 \vee \cdots \vee F_n$, where each F_i is a conjunction of literals.

In particular:

e.g. $\neg P \vee \neg Q$ is in NNF
$\neg(P \vee Q)$ is not in NNF

- A formula is in **negation normal form** (NNF) if and only if negations appears only in front of atoms.
- A **literal** is an atom or the negation of an atom.

- **Deduction theorem**: given a set of formulas, $\{F_1, \ldots, F_n\}$, and a formula G, $(F_1 \wedge \cdots \wedge F_n) \models G$ if and only if $\models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$. ($F_1 \wedge \ldots \wedge F_m \rightarrow G$ is valid)
- **Proof of refutation**: given a set of formulas, $\{F_1, \ldots, F_n\}$, and a formula G, $(F_1 \wedge \cdots \wedge F_n) \models G$ if and only if $F_1 \wedge \cdots \wedge F_n \wedge \neg G$ is inconsistent.

3.2.2 Proof Theory

Proof theory provides a formal way to prove logical consequences by syntactic manipulations of the symbols which appear in a given formula. Through syntactic manipulations it is possible to infer a specific conclusion, starting from some given premises. In particular, this process is called **natural deduction**. Moreover, in natural deduction, only the $\wedge$, $\rightarrow$ and $\bot$ symbols are used. In particular, some of the natural deduction rules are the following:

- **Introduction rules**:

label — premises — name of the rule

$$(\wedge I) \quad \frac{\varphi \quad \psi}{\varphi \wedge \psi} \wedge I$$

$\varphi \wedge \psi$: consequence

$$(\rightarrow I) \quad \frac{\begin{array}{c}[\varphi] \\ \vdots \\ \psi\end{array}}{\varphi \rightarrow \psi} \rightarrow I$$

if one assumes φ and one obtains ψ, then $\varphi \rightarrow \psi$. One can discard the assumption φ since it is contained in the consequence. Discarding φ is denoted with $[\varphi]$.

- **Elimination rules**:

$$(\wedge E) \quad \frac{\varphi \wedge \psi}{\varphi} \wedge E\,, \qquad \frac{\varphi \wedge \psi}{\psi} \wedge E \qquad\qquad (\rightarrow E) \quad \frac{\varphi \qquad \varphi \rightarrow \psi}{\psi} \rightarrow E$$

- **Ex falso sequitur quolibet**:

$$(\bot) \quad \frac{\bot}{\varphi} \bot$$

if one has $\bot$, then φ can assume any possible value.

- **Reductio ad absurdum**:

$$(RAA) \quad \frac{\begin{array}{c}[\neg\varphi] \\ \vdots \\ \bot\end{array}}{\varphi} RAA$$

if one starts from $\neg\varphi$ and obtains $\bot$ (false), then one obtains φ (dimostrazione per assurdo)

In particular:

- Each derivation defines a derivation tree, where the leafs contain the premises and the root is the conclusion.
- The notation $\Gamma \vdash \varphi$ means that there exists a derivation with premises Γ and conclusion φ. In case $\Gamma = \emptyset$, then φ is a theorem ($\vdash \varphi$).
- The notion of derivability ($\vdash$) and the notion of logical consequence ($\models$) should coincide:

$$\Gamma \vdash \varphi \Leftrightarrow \Gamma \models \varphi, \quad \text{where} \quad \begin{cases} \Gamma \vdash \varphi \Rightarrow \Gamma \models \varphi & \text{Soundness} \\ \Gamma \models \varphi \Rightarrow \Gamma \vdash \varphi & \text{Completeness} \end{cases}$$

Completeness theorem

Therefore, the set of theorems should coincide with the set of tautologies.

3.3 Resolution for Propositional Logic

In order to introduce resolution for propositional logic, it is necessary to define the following concepts:

- **Axioms**, which are given formulae, elementary tautologies and contradictions which can not be derived within the calculus.
- **Inference rules**, which allow to derive new formulae from given ones. An inference rule has the following form:

$$\frac{F_1 \quad \cdots \quad F_n}{F} \tag{3.1}$$

where the formulae $F_1, \ldots, F_n$ are the premises and the formula F is the conclusion.

- **Derivation** $\phi \vdash \psi$, which is a sequence of inference rule applications starting with formula ϕ and ending in formula ψ. A **derivation step** $F_1, \ldots, F_n \vdash F$ is the application of an inference rule.

Resolution in propositional logic is based on contradiction. In particular, the refutation theorem states that $\phi \models \psi$ if and only if $\not\models \phi \wedge \neg\psi$. The steps to apply resolution calculus in propositional logic are the following:

1. Transform all the formulae in conjunctive normal form (i.e. in clauses).
2. Apply the resolution inference rule.
3. Stop when the empty clause (i.e. $\square \equiv false$) is derived.

The following axiom and resolution rule to use are the following:

- **Axiom**: empty clause, $\square$.

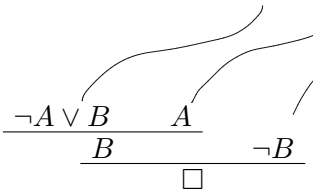
- **Resolution rule**:

$$\frac{R \vee A \qquad R' \vee \neg A}{R \vee R'}$$

  where R and R' are disjunctions of literals, A is an atomic formula and $R \vee R'$ is called *resolvent*.

For example, if one wants to prove that B is a logical consequence of $F = \{A \to B, A\}$, i.e. that $F \models B$ holds, one can proceed as follows:

1. F gets transformed into clausal form: $F' = \{\neg A \vee B, A\}$.
2. The negation of the conclusion is added to F': $F'' = \{\underline{\neg A \vee B}, \underline{A}, \underline{\neg B}\}$.
3. The resolution rule is applied:



4. At this point, one has obtained the empty clause, which means that F'' is inconsistent and therefore $F \models B$.

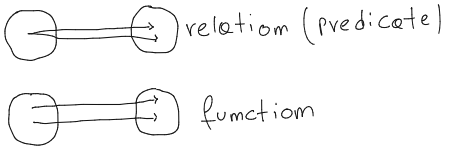
Chapter 4

First Order Logic

Propositional logic turns out to be too simple to make specific statements in specific domains. First order logic solves this problem by adding *quantifiers* and substituting propositions with *predicates* applied to *terms*.

4.1 Syntax

- The **signature** of a first order language is the following:



$$\Sigma = (\mathcal{P}, \mathcal{F}) \tag{4.1}$$

  where $\mathcal{P}$ is a finite set of predicates symbols and $\mathcal{F}$ is a finite set of function symbols.

- The first order logic **alphabet** contains the following:
  - $\mathcal{P}$, the set containing the predicate symbols, $p, q, r, \ldots$.
  - $\mathcal{F}$, the set of function symbols, $a, b, c, \ldots, f, g, h, \ldots$.
  - $\mathcal{V}$, the set of variables, $X, Y, Z, \ldots$.
  - Logic symbols:
    - Truth symbols, $\bot$ ($false$), $\top$ ($true$).
    - Logical connectives, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.
    - Quantifiers, $\forall$, $\exists$.
    - Syntactic symbols, (,), ,.

- The set of **terms**, $\mathcal{T}(\Sigma, \mathcal{V})$, is inductively defined as follows:
  - $X \in \mathcal{T}$ for all $X \in \mathcal{V}$.
  - $f(t_1, \ldots, t_n) \in \mathcal{T}$ for all $f \in \mathcal{F}$ with $t_1, \ldots, t_n \in \mathcal{T}$.

  For example, given the following elements: $a/0, f/1, g/2$ ($f/1$ is a function which takes one argument) X, a, $f(X)$, $g(f(X), g(Y, f(a)))$ are terms.

- The set of **well-formed formulae**, $\mathcal{F}(\Sigma, \mathcal{V}) = \{A, B, C, \dots\}$ is inductively composed of:
  - An atomic formula $p(\bar{t})$, for $p \in \mathcal{P}$ and $\bar{t} \in \mathcal{T}$. ($p(\bar{t}) = p(t_1, \dots, t_m)$, p is an m-ary predicate)
  - $\bot$ and $\top$.
  - $t_1 \doteq t_2$, for $t_1, t_2 \in \mathcal{T}$. – $\doteq$ is syntactic equality
  - $\neg F$, for $F \in \mathcal{F}$.
  - $F \wedge F'$, $F \vee F'$, $F \to F'$ for $F, F' \in \mathcal{F}$.
  - $\forall X F$ and $\exists X F$, for $X \in \mathcal{V}$ and $F \in \mathcal{F}$.

  For example:
  - Atomic formulae: $mortal(X)$, $mortal(socrates)$.
  - Non-atomic formulae: $mortal(socrates) \wedge mortal(father(socrates))$, $\forall X.mortal(X) \to mortal(father(X))$, $\exists X.X \doteq socrates$.

- Quantified formula $\forall X F$ and $\exists X F$ binds the variable X within the scope of F. In particular, the set of **free variables** of a given formula F, $\mathcal{F}_v$ is inductively defined as follows:
  - $\mathcal{F}_v(t_1 \doteq t_2) \coloneqq vars(t_1) \cup vars(t_2)$.
  - $\mathcal{F}_v(p(\bar{t})) \coloneqq \cup vars(\bar{t})$.
  - $\mathcal{F}_v(\top) \coloneqq \mathcal{F}_v \coloneqq \emptyset$.
  - $\mathcal{F}_v(\neg F) \coloneqq \mathcal{F}_v(F)$ for $F \in \mathcal{F}$.
  - $\mathcal{F}_v(F * F') \coloneqq \mathcal{F}_v(F) \cup \mathcal{F}_v(F')$ for $F, F' \in \mathcal{F}$ and $* \in \{\wedge, \vee, \to\}$.
  - $\mathcal{F}_v(\forall X F) \coloneqq \mathcal{F}_v(\exists X F) \coloneqq \mathcal{F}_v(F) \backslash \{X\}$ for $X \in \mathcal{V}$ and $F \in \mathcal{F}$.

  Moreover:
  - A **closed formula** or **sentence** is a formula which does not contain free variables.
  - A **theory** is a set of sentences.
  - A **ground term** is a formula which does not contain any variables.

4.2 Semantics

In first order logic, the presence of variables makes the task of defining a semantic more difficult than in the case of propositional logic. For example, the semantic of the formula $X \geq 1$ depends on the value of X, on the meaning of $\geq$ and on the value of the function (in this case constant) 1. In particular:

- An interpretation I of Σ consists of:
  - A universe U which is non-empty.
  - A function $I(f) : U^n \to U$ for every n-ary function symbol f of Σ.

– A relation, $I(p) \subseteq U^n$ for every n-ary predicate symbol p of Σ.

- The variable valuation for $\mathcal{V}$ with respect to I is represented by the following function:

$$\eta : \mathcal{V} \to U \text{ for every variable } X \in \mathcal{V} \text{ into the universe } U \text{ of } I \tag{4.2}$$

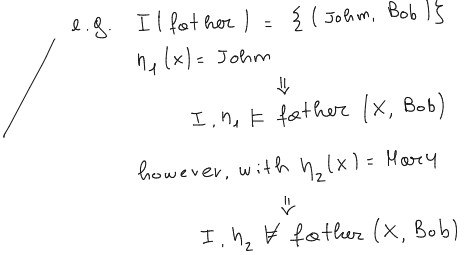
- The interpretation of terms is given by the following function:

$$\eta' : \mathcal{T}(\Sigma, \mathcal{V}) \to U \quad \Rightarrow \quad \begin{cases} \eta'(X) \coloneqq \eta(X) \\ \eta'(f(t_1, \ldots, t_n)) \coloneqq I(f)(\eta'(t_1), \ldots, \eta'(t_n)) \end{cases} \tag{4.3}$$

For example, given the signature $\Sigma = \{\emptyset, 1/0, */2, +/2\}$, the universe $U = \mathbb{N}$, the interpretations $I(1) \coloneqq 1$, $I(A \texttt{ + } B) \coloneqq A+B$ for $A, B \in U$, $I(A*B) \coloneqq AB$ for $A, B \in U$ and variable valuation $\eta(X) \coloneqq 3$, $\eta(Y) \coloneqq 5$:

$$\begin{aligned} \eta'(X * (Y \texttt{ + } 1)) &= I(*)(\eta'(X), \eta'(Y \texttt{ + } 1)) \\ &= I(*)(3, I(\texttt{+})(\eta'(Y), \eta'(1))) \\ &= \cdots = 3 \cdot (5 + 1) = 18 \end{aligned}$$

- Given I of Σ, η variable valuation, (I, η) satisfies a given formula F ($I, \eta \models F$) if:
  - $I, \eta \models \top$ and $I, \eta \not\models \bot$.
  - $I, \eta \models s \doteq t$ iff $\eta'(s) = \eta'(t)$.
  - $I, \eta \models p(t_1, \ldots, t_n)$ iff $(\eta'(t_1), \ldots, \eta'(t_n)) \in I(p)$.
  - $I, \eta \models \neg F$ iff $I, \eta \not\models F$.
  - $I, \eta \models F \wedge F'$ iff $I, \eta \models F$ and $I, \eta \models F'$.
  - $I, \eta \models F \vee F'$ iff $I, \eta \models F$ or $I, \eta \models F'$.
  - $I, \eta \models F \to F'$ iff $I, \eta \not\models F$ or $I, \eta \models F'$.
  - $I, \eta \models \forall X F$ iff $I, \eta[X \mapsto u] \models F$ for all $u \in U$.
  - $I, \eta \models \exists X F$ iff $I, \eta[X \mapsto u] \models F$ for some $u \in U$.



- Given a signature Σ, an interpretation I and a formula F:
  - I is a model of F or I satisfies F ($I \models F$) when $I, \eta \models F$ for every variable valuation η.
  - I is a model of theory T when I is a model of each formula in T.

- Given a sentence S:
  - S is **valid** if it is satisfied by every interpretation, $I \models S$ for every I.
  - S is **satisfiable** if it is satisfied by some interpretations, $I \models S$ for some I.

- S is **falsifiable** if it is not satisfied by some interpretations, $I \not\models S$ for some I.
- S is **unsatisfiable** if it is not satisfied by any interpretation, $I \not\models S$ for every I.

- A sentence or theory T_1 is a **logical consequence** of a sentence or theory T_2, $T_2 \models T_1$, if every model of T_2 is also a model of T_1, i.e. $I \models T_2$ implies $I \models T_1$.

  Moreover, two sentences or theories are **equivalent** ($\Leftrightarrow$) if they are logical consequences of each other.

4.3 Resolution for First Order Logic

Resolution in first order logic is based on contradiction as for propositional logic: theory united with negated consequence must be unsatisfiable. In order to prove so, one has to transform first order logic formulae in clauses. Finally one has to define the resolution inference rule for first order logic, and this last step requires substitutions and unification. In particular:

- For every sentence F, there exist an equivalent sentence, F_{neg}, in **negated normal form**:
  - $\neg\bot \Leftrightarrow \top, \neg\top \Leftrightarrow \bot$.
  - $\neg\neg F \Leftrightarrow F$.
  - $\neg(F \wedge F') \Leftrightarrow \neg F \vee \neg F', \neg(F \vee F') \Leftrightarrow \neg F \wedge \neg F'$.
  - $\neg\forall X F \Leftrightarrow \exists X \neg F, \neg\exists X F \Leftrightarrow \forall X \neg F$.
  - $\neg(F \rightarrow F') \Leftrightarrow F \wedge \neg F'$.

- For every sentence F, there exist an equivalent sentence in **prenex form** of the form:

$$F = Q_1 X_1, \ldots, Q_n X_n G \tag{4.4}$$

  In particular, Q_i are the quantifiers, X_i are the variables and G is a formula without quantifiers. For example:

$$\begin{aligned}\neg\exists x p(x) \vee \forall x r(x) &\Leftrightarrow \forall x \neg p(x) \vee \forall x r(x)\\ &\Leftrightarrow \forall x \neg p(x) \vee \forall y r(y)\\ &\Leftrightarrow \forall x(\neg p(x) \vee \forall y r(y))\\ &\Leftrightarrow \forall x \forall y(\neg p(x) \vee r(y))\end{aligned}$$

- Given a formula, $F \in \mathcal{F}(\Sigma, \mathcal{V})$, in negation normal form and a function (**Skolem function**), f/n, not occurring in Σ, the **Skolemized form** of F, F', is obtained by replacing $\exists X G$ with $G[X \mapsto f(\bar{V})]$ in F, where $\bar{V}$ are the free variables in the sub-formula G of F. For example:

$$\forall z \exists x \forall y(p(x,z) \wedge q(g(x,y),x,z)) \Leftrightarrow \forall z \forall y(p(f(z),z) \wedge q(g(f(z),y),f(z),z))$$

- A **clause** (in disjunctive normal form) is a formula of the form:

$$\bigvee_{i=1}^{n} L_i \tag{4.5}$$

where L_i are literals (atom or negation of atom). Moreover, a formula of the form:

$$F = \bigwedge_{j=1}^{n} B_j \rightarrow \bigvee_{k=1}^{m} H_k \tag{4.6}$$

can be re-written as:

$$F = \bigvee_{i=1}^{n+m} L_i, \quad \text{with } L_i = \begin{cases} \neg B_i & \text{for } i = 1, \ldots, n \\ H_{i-n} & \text{i = n + 1, ..., n + m} \end{cases} \tag{4.7}$$

where B_j and H_k are atoms.

- A **closed clause** is a sentence of the form $\forall \bar{x} C$, where C is a clause.
- A **clausal form of theory** consists of closed clauses.
- An arbitrary theory T can be transformed into clausal form as follows:
  - Convert every formula in the theory into an equivalent formula in negated normal form.
  - Perform Skolemization in order to eliminate all existential quantifiers.
  - Convert the resulting theory, which is still in negated normal form, into an equivalent theory in clausal form, i.e. move conjunctions and universal quantifiers outwards.

substitution allows injective renaming (mapping from variables to variables) However, $x \mapsto z, y \mapsto z$ is not a renaiming (not injective)

- A **substitution** $\sigma : \mathcal{V} \rightarrow \mathcal{T}(\Sigma, \mathcal{V}')$ is a mapping which modifies a finite number of variables. For example, $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$, where each X_i is a distinct variable and each t_i is a term. The identity substitution is indicated as $\epsilon = \emptyset$. Substitution can be also applied to terms: $\sigma : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathcal{T}(\Sigma, \mathcal{V}')$, i.e. $f(\bar{t})\sigma \coloneqq f(t_1\sigma, \ldots, t_n\sigma)$.
- A **logical expression** (or *expression*) over the set $\mathcal{V}$ is inductively composed by:
  - Terms with variables in $\mathcal{V}$.
  - Formulae with free variables in $\mathcal{V}$.
  - Substitutions from an arbitrary set of variables into $\mathcal{T}(\Sigma, \mathcal{V})$.
  - Tuples of logical expression over $\mathcal{V}$.

e.g. $\sigma = \{x \mapsto z, y \mapsto 5\}$: $(x \cdot (y+1))\sigma = z \cdot (5+1)$

  Moreover, a logical expression is a **simple expression** if it does not contain quantifiers.
- An expression e is an **instance** of e' iff $e = e'\sigma$. In this case, e' is said to be more general than e. Moreover, **variable renaming** for an expression e is a substitution σ where:

- σ is injective.
- $X\sigma \in \mathcal{V}\ \forall X \in \mathcal{V}$.
- $X\sigma$ does not occur in e for free variables X of e.

Moreover, the expression e and e' are said to be **variants** iff $e = e'\sigma$ and $e' = e\tau$.

syntactic equality

- A specific substitution σ is said to be a **unifier** for the simple expressions $e_1, \ldots, e_n$ if $e_1\sigma = \cdots = e_n\sigma$. Moreover, σ is said to be the **most general unifier** (MGU) for $e_1, \ldots, e_n$ iff every unifier τ for $e_1, \ldots, e_n$ is ~~instance of~~ derived from σ, namely $\tau = \sigma\rho$ for some ρ. For example:

$$f(X, a) \doteq f(g(U), Y) \doteq Z \quad \stackrel{\text{MGU}}{\Longrightarrow} \quad \sigma = \{X \mapsto g(U), Y \mapsto a, Z \mapsto f(g(U), a)\}$$

As for propositional logic, resolution in first order logic works by contradiction. In particular:

- The only axiom is the empty clause, $\square$.
- The resolution rule is the following:

$$\frac{R \vee A \qquad R' \vee \neg A'}{(R \vee R')\sigma}$$

where σ is the most general unifier for the atomic formulae A and A'. It is important that $R \vee A$ and $R' \vee \neg A'$ have different variables.

As for propositional logic, assuming that $\vdash$ denotes resolution and F is a set of clauses, in first order logic:

$$F \vdash \bot \text{ iff } F \models \bot$$

However, it is undecidable whether a first order logic formula is provable, namely *true* under all possible interpretations.

Chapter 5

Logic Programming Languages

In logic programming languages, programs and queries are logic formulas. In particular, one can consider them as a first order logic theory and provide a logical reading in terms of:

- Model theory (declarative semantics).
- Proof theory (operational semantics).
- Soundness and completeness.

In logic programming languages, computation is exactly the same as logic deduction (resolution). This allows to prove properties of programs (e.g. correctness). Moreover, logic programming languages are declarative languages, in the sense that the programmer only specifies what to do, while the machine takes care of how to infer the conclusion.

5.1 Syntax

The syntax of logic programming languages is provided by the **extended Backus-Naur form** (EBNF) notation, which is a notation consisting of production rules of the form:

$$Name\colon\ G, H ::= A \mid B,\ Condition \tag{5.1}$$

where A and B are syntactical entities, G and H are defined by the rule $Name$. In particular, if $Condition$ holds, then G and H can be of the form A or B. In particular, logic programming calculus uses the following structures:

predicate terms

$$\begin{array}{lrcl} Atom\colon & A, B & ::= & p(t_1, \ldots, t_n),\ n \geq 0 \\ Goal\colon & G, H & ::= & \top \mid \bot \mid A \mid G \wedge H \\ Clause\colon & K & ::= & A \leftarrow G \text{ — body} \\ Program\colon & P & ::= & K_1, \ldots, K_m,\ m \geq 0 \end{array}$$

head

5.2 Semantics

The semantics of logic programming languages is provided in terms of *state transition systems*. In particular:

- A simple **state transition system** (TS) is a pair $\langle S, \mapsto \rangle$ where:
  - S is a set of states which includes initial and final states.
  - $\mapsto$ is a binary (transition) relation over states. The transition from S_1 to S_2, $S_1 \mapsto S_2$, is possible if some given condition holds. In particular, the $\mapsto$ relation is defined by using transition rules of the form:

    $$\begin{array}{ll} \text{IF} & \mathit{Condition} \\ \text{THEN} & S_1 \mapsto S_2 \end{array}$$

  - The consecutive application of the $\mapsto$ relation, $S_1 \mapsto S_2 \mapsto \cdots \mapsto S_n$ is called a **derivation**.
  - The relation $\mapsto^*$ is the reflexive-transitive closure of $\mapsto$.
- A **state** is a pair $\langle G, \theta \rangle$ where G is a goal and θ is a substitution. In particular:
  - An initial state has the form $\langle G, \epsilon \rangle$.
  - A successful final state has the form $\langle \top, \theta \rangle$.
  - A failed final state has the form $\langle \bot, \epsilon \rangle$.
- A **derivation** is said to be:
  - Successful if its final state is successful.
  - Failed if its final state is failed.
  - Infinite is there exists an infinite sequence of states and transitions $S_1 \mapsto S_2, \mapsto S_3 \mapsto \ldots$.
- A **goal** is said to be:
  - Successful is it has a successful derivation starting with $\langle G, \epsilon \rangle$.
  - Finitely failed is it has only failed derivations starting with $\langle G, \epsilon \rangle$.
- Given an initial goal (query) G and a program P, if there exists a successful derivation in P such that:

  $$\langle G, \epsilon \rangle \mapsto^* \langle \top, \theta \rangle \tag{5.2}$$

  then the substitution θ is called **computed answer substitution** (CAS) of G in P. In particular, the computed answer substitution is the result of the computation, namely the values associated by the computed answer substitution to the variables in the goal provide the results.

The main transition rules which describe the semantics of logic programming are the following:

- The **unfold** rule:

IF	$(H \leftarrow B)$ is a fresh variant of a clause in P
AND	β is the most general unifier of H and $A\theta$
THEN	$\langle A \wedge G, \theta \rangle \mapsto \langle B \wedge G, \theta\beta \rangle$

Program (set of clauses)

if MGU is found, then the body is evaluated

- The **failure** rule:

IF	there is no clause $(H \leftarrow B)$ in P
WITH	a unifier of H and $A\theta$
THEN	$\langle A \wedge G, \theta \rangle \mapsto \langle \bot, \epsilon \rangle$

In can be observed that the unfold rule exhibits two kinds of non-determinism:

- The **don't-care non-determinism**, by which any atom in $A \wedge G$ can be chosen as the atom A. This basically affects the length of the derivation process.
- The **don't-know non-determinism**, by which any clause $H \leftarrow B$ in P, for which H and $A\theta$ are unifiable, can be chosen. This determines the computed answer substitution of the derivation process.

In order to avoid this, the **Selective Linear Definite clause resolution** (SLD resolution) can be applied:

Prolog selects the leftmost atom and applies depth-first search

- This resolution process uses a selection function to select the atom in the goal for the application of the unfold rule. This eliminates the don't-care non-determinism.
- By using all the possible clauses with a specific selection rule one obtains a search tree called **SLD tree**. Therefore, in order to eliminate also the don't-know non-determinism, one must define a search strategy.

Moreover, given a logic program P, a goal G and a substitution θ:

- If θ is a computed answer substitution of G, then $P \models G\theta$ (soundness).
- If $P \models G\theta$, then a computed answer substitution σ, more general than θ, of G exists such that $G\theta = G\sigma\beta$ (completeness).
- If a computed answer substitution σ of G can be obtained by using a specific selection rule r, then it can be obtained also by using a different selection rule r'.

Chapter 6

Prolog

Prolog is a declarative logic programming language in which the program logic is expressed in terms of relations, represented as facts and rules and where a computation is initiated by running a query over these relations.

6.1 Syntax

Prolog uses the following syntax:

Variables:	`X`, `Y`, `_`	with capital initial letter.
Constants:	`alice`, `a`	with lower initial letter.
Functions:	`father(X)`	with lower initial letter.
Predicates:	`brother(X, Y)`	with lower initial letter.
Lists:	`[]`, `[t1 | t2]`	where `[]` is the empty list and `[t1 | t2]` is the list whose first element (head) is `t1` and the rest of the list (tail) is `t2`.
Facts:	`head.`	where `head` is a conjunction of literals.
Rules:	`head :- body.`	where `head` and `body` are conjunctions of literals.

Moreover, Prolog is not typed, namely there exists no types.

6.2 Semantics

When a query is executed, Prolog applies SLD resolution. In particular:

- As selection rule, Prolog always selects the leftmost atom in the goal.
- As search strategy, Prolog always selects clauses according to the syntactical ordering provided for the clauses.

The search process can be stopped by using the predicate `!` (cut). In particular, during this search process, one can encounter repeated paths and failed paths and this is source of inefficiency. The operator `!` allows to prune part of the useless paths.

6.3 Arithmetic

In Prolog, integers, floats and numbers of a given base x are pre-defined constants. For example:

Integers:	`0`, `1`, `9977`, `-79393`
Floats:	`4.5e3`, `1.0`, `-0.5e7`
Base x:	`2'101`, `8'174`

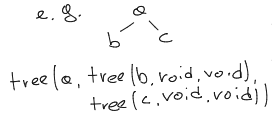
Moreover:

- There exists pre-defined numerical functions. For example: `+`, `-`, `*`, `abs`, `max`, and others.
- There exist pre-defined predicates. For example:

`Z is X`	the value of `X` is unified with `Z`.
`X =:= Y`	the value of `X` and `Y` are equal.
`X =\= Y`	the value of `X` and `Y` are different.
`X > Y`	the value of `X` is greater than the value of `Y`.

6.4 Data Structures

In Prolog, data structures are represented by using terms. For example, considering a binary tree:



`void`	represents the empty tree.
`tree(Root, Left, Right)`	represents a non-empty tree.

Chapter 7

Constraint Programming Languages

In general, there exist two families of constraint programming languages:

- **Constraint Logic Programming** languages (CLPs), where constraints, and the relative solvers, are added to logic programming. These type of languages are more expressive, flexible and more efficient than simple logic programs. For example: Prolog with comstraints.
- **Imperative languages with constrains**, where constraints, and the relative solvers, are added to imperative languages. For example: MiniZinc.

In particular:

- A **constraint** is a conjunction of predicates. For example: $4X + 3Y = 10 \land 2X - Y = 0$.
- A **constraint problem** (query) is represented by an initial constraint. For example: $X = Y?$.
- A **constraint solution** (answer) is a valuation for the problem's variables which satisfies all the given constraints. For example: $X = 1 \land Y = 2$.
- A program is declaratively modelled by using constraints, which give a description of properties and relationships between partially known objects.
- Constraints allow to introduce the following concepts:
  - **Propagation** of the effects of new information (in form of constraints).
  - **Simplification**, which makes implicit information explicit.

  These concepts are then used to reduce the overall search space.

In general, there exist two classes of constraint problems:

- **Constraint Satisfaction Problems** (CSPs), which are defined by:
  - A finite set of variables $\{X_1, \ldots, X_n\}$.

– A domain for each variable $D(X_1), \ldots, D(X_n)$.
– A set of constraints $\{C_1, \ldots, C_n\}$.

A solution to these type of problems is a complete assignment to all the variables which satisfies the constraints.

- **Constraint Optimization Problems** (COPs), which are CSPs defined of the variables $\{X_1, \ldots, X_n\}$ and domains $D(X_1), \ldots, D(X_n)$, together with an objective function $f : D(X_1), \ldots, D(X_n) \to D$. A solution to these type of problems is a solution to the corresponding CSPs which optimize the value of f.

7.1 Constraint Logic Programming

As for logic programming, it is necessary to define a syntax and an operational semantics.

7.1.1 Syntax

Constraint logic programming syntax uses the following structures:

Atom:	A, B	$::=$	$p(t_1, \ldots, t_n)$, $n \geq 0$
Constraint:	C, D	$::=$	$c(t_1, \ldots, t_n) \mid C \wedge D$, $n \geq 0$
Goal:	G, H	$::=$	$\top \mid \bot \mid A \mid C \mid G \wedge H$
CL clause:	K	$::=$	$A \leftarrow G$
CL program:	P	$::=$	$K_1, \ldots, K_m$, $m \geq 0$

7.1.2 Semantics

Constraint logic programming semantics is defined using transition systems. In particular:

- Each state of the system is a pair $\langle G, C \rangle$, where G is a goal and C a constraint.
- The initial state of the system is $\langle G, \top \rangle$.
- A successful final state of the system is $\langle \top, C \rangle$, where $C \neq \bot$.
- A failed final state of the system is $\langle G, \bot \rangle$.

The main transition rules which describe the semantics of constraint logic programming are the following:

- The **unfold** rule:

$$\begin{array}{ll} \text{IF} & (H \leftarrow B) \text{ is a fresh variant of a clause in } P \\ \text{AND} & CT \models \exists((H \doteq A) \wedge C) \\ \text{THEN} & \langle A \wedge G, C \rangle \mapsto \langle B \wedge G, (H \doteq A) \wedge C \rangle \end{array}$$

As for logic programming, even in constraint logic programming there exist non-determinism in the choice of the specific goal and in the selection of the specific clause.

- The **failure** rule:

IF	there is no clause $(H \leftarrow B)$ in P
WITH	$CT \models \exists((H \doteq A) \wedge C)$
THEN	$\langle A \wedge G, C\rangle \mapsto \langle \bot, \bot\rangle$

- The **solve** rule:

allows simplification
e.g. $x=0 \wedge y=x \wedge y<10$ $\downarrow$ $x=0 \wedge y=x$

IF	$CT \models \forall((C \wedge D_1) \leftrightarrow D_2)$
THEN	$\langle C \wedge G, D_1\rangle \mapsto \langle G, D_2\rangle$

In order to simplify computation, one needs to specify constraints and them to label variables.

Chapter 8

MiniZinc

MiniZinc is a free and open-source constraint modelling language. This language can be used to model constraint satisfaction and optimization problems in a high-level, solver-independent way, taking advantage of a large library of pre-defined constraints. In particular, a specific model is then compiled into FlatZinc, a solver input language which is understood by a wide range of solvers.

8.1 Syntax

The main syntactical items of MiniZinc are the following:

- Inclusion of libraries:

```
include "[library name];"
```

- Definition of parameters: — its value will be given by the problem instance (equivalent to any variable in, e.g., C)

```
[domain]: [parameter name];
```

- Definition of variables: – its domain values will be checked in order to find a solution (the assignment is done by the solver)

```
var [domain]: [variable name];
```

- Definition of arrays:

```
array[index domain] of [domain];,
array[index domain] of var [domain];
```

- Definition of constraints:

```
constraint [expression];
```

e.g.: Travelling Salesman Problem

```
include "globals.mzn";

int: n;
array [1..n,1..n] of int dist;
int: start_city,
int: end_city;
array [1..n] of var 1..n: city;
array[1..n] of string: city_name;

constraint city[1] = start_city;
constraint city[n] = end_city;
constraint all_different(city);
```

- Usage of aggregation functions:

```
sum,
product,
min,
max,
forall([counter(s)] in [domain]) ([constraints]);,
forall([counter(s)] in [domain] where [condition]) ([constraints]);,
exists([counter(s)] in [domain]) ([constraints]);,
...
```

- Usage of arithmetic operators: `+`, `-`, `*`, `/`, `^`, `=`, `!=`.
- Usage of logical operators: `\/` (logical or), `/\` (logical and), `->` (logical implication), `!` (logical negation).

8.2 Usage

In order to find a solution to a CSP one has to call the solver to find a single solution. In particular, this is achieved by adding the following call:

```
solve satisfy;
```

While, for solving a COP, one has to first define the objective and the aim of the solution as follows:

```
var [domain]: [variable name] = [aim of solution];
```

and then has to add the following call:

```
solve minimize [objective];
```