

# Languages and Algorithms for Artificial Intelligence (Module 2)

Last update: 22 December 2023

Academic Year 2023 – 2024  
Alma Mater Studiorum · University of Bologna

# Contents

<b>1</b>	<b>Propositional logic</b>	<b>1</b>
1.1	Syntax . . . . .	1
1.2	Semantics . . . . .	1
1.2.1	Normal forms . . . . .	2
1.3	Reasoning . . . . .	3
1.3.1	Natural deduction . . . . .	4
<b>2</b>	<b>First order logic</b>	<b>5</b>
2.1	Syntax . . . . .	5
2.2	Semantics . . . . .	6
2.3	Substitution . . . . .	6
<b>3</b>	<b>Logic programming</b>	<b>8</b>
3.1	Syntax . . . . .	8
3.2	Semantics . . . . .	8
3.2.1	State transition system . . . . .	8
<b>4</b>	<b>Prolog</b>	<b>10</b>
4.1	Syntax . . . . .	10
4.2	Semantics . . . . .	11
4.3	Arithmetic operators . . . . .	11
4.4	Lists . . . . .	12
4.5	Cut . . . . .	12
4.6	Negation . . . . .	13
4.7	Meta predicates . . . . .	14
4.8	Meta-interpreters . . . . .	16
<b>5</b>	<b>Constraint programming</b>	<b>18</b>
5.1	Constraint logic programming (CLP) . . . . .	18
5.1.1	Syntax . . . . .	18
5.1.2	Semantics . . . . .	18
5.2	MiniZinc . . . . .	19

# 1 Propositional logic

## 1.1 Syntax

**Syntax** Rules and symbols to define well-formed sentences.

Syntax

The symbols of propositional logic are:

**Proposition symbols**  $p_0, p_1, \dots$

**Connectives**  $\wedge \vee \rightarrow \leftrightarrow \neg \perp ( )$

**Well-formed formula** The definition of a well-formed formula is recursive:

Well-formed formula

- An atomic proposition is a well-formed formula.
- If  $S$  is well-formed,  $\neg S$  is well-formed.
- If  $S_1$  and  $S_2$  are well-formed,  $S_1 \wedge S_2$  is well-formed.
- If  $S_1$  and  $S_2$  are well-formed,  $S_1 \vee S_2$  is well-formed.

Note that the implication  $S_1 \rightarrow S_2$  can be written as  $\neg S_1 \vee S_2$ .

The BNF definition of a formula is:

$$F := \text{atomic\_proposition} \mid F \wedge F \mid F \vee F \mid F \rightarrow F \mid F \leftrightarrow F \mid \neg F \mid (F)$$

## 1.2 Semantics

**Semantics** Rules to associate a meaning to well-formed sentences.

Semantics

**Model theory** What is true.

**Proof theory** What is provable.

**Interpretation** Given a propositional formula  $F$  of  $n$  atoms  $\{A_1, \dots, A_n\}$ , an interpretation  $\mathcal{I}$  of  $F$  is a pair  $(D, I)$  where:

Interpretation

- $D$  is the domain. Truth values in the case of propositional logic.
- $I$  is the interpretation mapping that assigns to the atoms  $\{A_1, \dots, A_n\}$  an element of  $D$ .

Note: given a formula  $F$  of  $n$  distinct atoms, there are  $2^n$  distinct interpretations.

**Model** If  $F$  is true under the interpretation  $\mathcal{I}$ , we say that  $\mathcal{I}$  is a model of  $F$  ( $\mathcal{I} \models F$ ).

Model

**Valid formula** A formula  $F$  is valid (tautology) iff it is true in all the possible interpretations. It is denoted as  $\models F$ .

Valid formula

**Invalid formula** A formula  $F$  is invalid iff it is not valid ( $\not\models$ ).

Invalid formula

In other words, there is at least an interpretation where  $F$  is false.

**Inconsistent formula** A formula  $F$  is inconsistent (unsatisfiable) iff it is false in all the possible interpretations. Inconsistent formula

**Consistent formula** A formula  $F$  is consistent (satisfiable) iff it is not inconsistent. Consistent formula  
In other words, there is at least an interpretation where  $F$  is true.

**Decidability** A logic is decidable if there is a terminating method to decide if a formula is valid. Decidability  
Propositional logic is decidable.

**Truth table** Useful to define the semantics of connectives. Truth table

- $\neg S$  is true iff  $S$  is false.
- $S_1 \wedge S_2$  is true iff  $S_1$  is true and  $S_2$  is true.
- $S_1 \vee S_2$  is true iff  $S_1$  is true or  $S_2$  is true.
- $S_1 \rightarrow S_2$  is true iff  $S_1$  is false or  $S_2$  is true.
- $S_1 \leftrightarrow S_2$  is true iff  $S_1 \rightarrow S_2$  is true and  $S_1 \leftarrow S_2$  is true.

**Evaluation** The connectives of a propositional formula are evaluated in the order: Evaluation order

$$\leftrightarrow, \rightarrow, \vee, \wedge, \neg$$

Formulas in parenthesis have higher priority.

**Logical consequence** Let  $\Gamma = \{F_1, \dots, F_n\}$  be a set of formulas (premises) and  $G$  a formula (conclusion).  $G$  is a logical consequence of  $\Gamma$  ( $\Gamma \models G$ ) if in all the possible interpretations  $\mathcal{I}$ , if  $F_1 \wedge \dots \wedge F_n$  is true,  $G$  is true. Logical consequence

**Logical equivalence** Two formulas  $F$  and  $G$  are logically equivalent ( $F \equiv G$ ) iff the truth values of  $F$  and  $G$  are the same under the same interpretation. In other words,  $F \equiv G \iff F \models G \wedge G \models F$ . Logical equivalence

Common equivalences are:

**Commutativity** :  $(P \wedge Q) \equiv (Q \wedge P)$  and  $(P \vee Q) \equiv (Q \vee P)$

**Associativity** :  $((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$  and  $((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$

**Double negation elimination** :  $\neg(\neg P) \equiv P$

**Contraposition** :  $(P \rightarrow Q) \equiv (\neg Q \rightarrow \neg P)$

**Implication elimination** :  $(P \rightarrow Q) \equiv (\neg P \vee Q)$

**Biconditional elimination** :  $(P \leftrightarrow Q) \equiv ((P \rightarrow Q) \wedge (Q \rightarrow P))$

**De Morgan** :  $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$  and  $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$

**Distributivity of  $\wedge$  over  $\vee$**  :  $(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$

**Distributivity of  $\vee$  over  $\wedge$**  :  $(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R))$

### 1.2.1 Normal forms

**Negation normal form (NNF)** A formula is in negation normal form iff negations appear only in front of atoms (i.e. not parenthesis). Negation normal form

**Conjunctive normal form (CNF)** A formula  $F$  is in conjunctive normal form iff: Conjunctive normal form

- it is in negation normal form;

- it has the form  $F := F_1 \wedge F_2 \cdots \wedge F_n$ , where each  $F_i$  (clause) is a disjunction of literals.

**Example.**

$(\neg P \vee Q) \wedge (\neg P \vee R)$  is in CNF.

$\neg(P \vee Q) \wedge (\neg P \vee R)$  is not in CNF (not in NNF).

**Disjunctive normal form (DNF)** A formula  $F$  is in disjunctive normal form iff:

Disjunctive normal form

- it is in negation normal form;
- it has the form  $F := F_1 \vee F_2 \cdots \vee F_n$ , where each  $F_i$  is a conjunction of literals.

## 1.3 Reasoning

**Reasoning method** Systems to work with symbols.

Reasoning method

Given a set of formulas  $\Gamma$ , a formula  $F$  and a reasoning method  $E$ , we denote with  $\Gamma \vdash^E F$  the fact that  $F$  can be deduced from  $\Gamma$  using the reasoning method  $E$ .

**Sound** A reasoning method  $E$  is sound iff:

Soundness

$$(\Gamma \vdash^E F) \rightarrow (\Gamma \models F)$$

**Complete** A reasoning method  $E$  is complete iff:

Completeness

$$(\Gamma \models F) \rightarrow (\Gamma \vdash^E F)$$

**Deduction theorem** Given a set of formulas  $\{F_1, \dots, F_n\}$  and a formula  $G$ :

Deduction theorem

$$(F_1 \wedge \cdots \wedge F_n) \models G \iff \models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$$

*Proof.*

$\rightarrow$  ) By hypothesis  $(F_1 \wedge \cdots \wedge F_n) \models G$ .

So, for each interpretation  $\mathcal{I}$  in which  $(F_1 \wedge \cdots \wedge F_n)$  is true,  $G$  is also true. Therefore,  $\mathcal{I} \models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$ .

Moreover, for each interpretation  $\mathcal{I}'$  in which  $(F_1 \wedge \cdots \wedge F_n)$  is false,  $(F_1 \wedge \cdots \wedge F_n) \rightarrow G$  is true. Therefore,  $\mathcal{I}' \models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$ .

In conclusion,  $\models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$ .

$\leftarrow$  ) By hypothesis  $\models (F_1 \wedge \cdots \wedge F_n) \rightarrow G$ . Therefore, for each interpretation where  $(F_1 \wedge \cdots \wedge F_n)$  is true,  $G$  is also true.

In conclusion,  $(F_1 \wedge \cdots \wedge F_n) \models G$ .

□

**Refutation theorem** Given a set of formulas  $\{F_1, \dots, F_n\}$  and a formula  $G$ :

Refutation theorem

$$(F_1 \wedge \cdots \wedge F_n) \models G \iff F_1 \wedge \cdots \wedge F_n \wedge \neg G \text{ is inconsistent}$$

Note: this theorem is not accepted in intuitionistic logic.

*Proof.* By definition,  $(F_1 \wedge \cdots \wedge F_n) \models G$  iff for every interpretation where  $(F_1 \wedge \cdots \wedge F_n)$  is true,  $G$  is also true. This requires that there are no interpretations where  $(F_1 \wedge \cdots \wedge F_n)$  is true and  $G$  false. In other words, it requires that  $(F_1 \wedge \cdots \wedge F_n \wedge \neg G)$  is inconsistent. □

### 1.3.1 Natural deduction

**Proof theory** Set of rules that allows to derive conclusions from premises by exploiting syntactic manipulations. Proof theory

**Natural deduction** Set of rules to introduce or eliminate connectives. We consider a subset  $\{\wedge, \rightarrow, \perp\}$  of functionally complete connectives. Natural deduction for propositional logic

Natural deduction can be represented using a tree like structure:

$$\begin{array}{c} [\text{hypothesis}] \\ \vdots \\ \frac{\text{premise}}{\text{conclusion}} \text{ rule name} \end{array}$$

The conclusion is true when the hypothesis are able to prove the premise. Another tree can be built on top of premises to prove them.

**Introduction** Usually used to prove the conclusion by splitting it. Introduction rules

$$\begin{array}{c} \frac{\psi \quad \varphi}{\varphi \wedge \psi} \wedge\text{I} \qquad \begin{array}{c} [\varphi] \\ \vdots \\ \frac{\psi}{\varphi \rightarrow \psi} \rightarrow\text{I} \end{array} \end{array}$$

**Elimination** Usually used to exploit hypothesis and derive a conclusion. Elimination rules

$$\frac{\varphi \wedge \psi}{\varphi} \wedge\text{E} \qquad \frac{\varphi \wedge \psi}{\psi} \wedge\text{E} \qquad \frac{\varphi \quad \varphi \rightarrow \psi}{\psi} \rightarrow\text{E}$$

**Ex falso sequitur quodlibet** From contradiction, anything follows. This can be used when we have two contradicting hypothesis. Ex falso sequitur quodlibet

$$\frac{\perp}{\varphi} \perp$$

**Reductio ad absurdum** Assume the opposite and prove a contradiction (not accepted in intuitionistic logic). Reductio ad absurdum

$$\begin{array}{c} [\neg\varphi] \\ \vdots \\ \frac{\perp}{\varphi} \text{RAA} \end{array}$$

## 2 First order logic

### 2.1 Syntax

The symbols of propositional logic are:

Syntax

**Constants** Known elements of the domain. Do not represent truth values.

**Variables** Unknown elements of the domain. Do not represent truth values.

**Function symbols** Function  $f^{(n)}$  applied on  $n$  constants to obtain another constant.

**Predicate symbols** Function  $P^{(n)}$  applied on  $n$  constants to obtain a truth value.

**Connectives**  $\forall \exists \wedge \vee \rightarrow \neg \leftrightarrow \top \perp ( )$

Using the basic syntax, the following constructs can be defined:

**Term** Denotes elements of the domain.

$$t := \text{constant} \mid \text{variable} \mid f^{(n)}(t_1, \dots, t_n)$$

**Proposition** Denotes truth values.

$$P := \top \mid \perp \mid P \wedge P \mid P \vee P \mid P \rightarrow P \mid P \leftrightarrow P \mid \neg P \mid \forall x.P \mid \exists x.P \mid (P) \mid P^{(n)}(t_1, \dots, t_n)$$

**Well-formed formula** The definition of well-formed formula in first order logic extends the one of propositional logic by adding the following conditions:

Well-formed formula

- If  $S$  is well-formed,  $\exists X.S$  is well-formed. Where  $X$  is a variable.
- If  $S$  is well-formed,  $\forall X.S$  is well-formed. Where  $X$  is a variable.

**Free variables** The universal and existential quantifiers bind their variable within the scope of the formula. Let  $F_v(F)$  be the set of free variables in a formula  $F$ ,  $F_v$  is defined as follows:

Free variables

- $F_v(p(t)) = \bigcup \text{vars}(t)$
- $F_v(\top) = F_v(\perp) = \emptyset$
- $F_v(\neg F) = F_v(F)$
- $F_v(F_1 \wedge F_2) = F_v(F_1 \vee F_2) = F_v(F_1 \rightarrow F_2) = F_v(F_1) \cup F_v(F_2)$
- $F_v(\forall X.F) = F_v(\exists X.F) = F_v(F) \setminus \{X\}$

**Closed formula/Sentence** Proposition without free variables.

Sentence

**Theory** Set of sentences.

Theory

**Ground term/Formula** Proposition without variables.

Formula

## 2.2 Semantics

**Interpretation** An interpretation in first order logic  $\mathcal{I}$  is a pair  $(D, I)$ :

Interpretation

- $D$  is the domain of the terms.
- $I$  is the interpretation function such that:
  - $I(f) : D^n \rightarrow D$  for every  $n$ -ary function symbol.
  - $I(p) \subseteq D^n$  for every  $n$ -ary predicate symbol.

**Variable evaluation** Given an interpretation  $\mathcal{I} = (D, I)$  and a set of variables  $\mathcal{V}$ , a variable is evaluated through  $\eta : \mathcal{V} \rightarrow D$ .

Variable evaluation

**Model** Given an interpretation  $\mathcal{I}$  and a formula  $F$ ,  $\mathcal{I}$  models  $F$  ( $\mathcal{I} \models F$ ) when  $\mathcal{I}, \eta \models F$  for every variable evaluation  $\eta$ .

Model

A sentence  $S$  is:

**Valid**  $S$  is satisfied by every interpretation ( $\forall \mathcal{I} : \mathcal{I} \models S$ ).

**Satisfiable**  $S$  is satisfied by some interpretations ( $\exists \mathcal{I} : \mathcal{I} \models S$ ).

**Falsifiable**  $S$  is not satisfied by some interpretations ( $\exists \mathcal{I} : \mathcal{I} \not\models S$ ).

**Unsatisfiable**  $S$  is not satisfied by any interpretation ( $\forall \mathcal{I} : \mathcal{I} \not\models S$ ).

**Logical consequence** A sentence  $T_1$  is a logical consequence of  $T_2$  ( $T_2 \models T_1$ ) if every model of  $T_2$  is also model of  $T_1$ :

Logical consequence

$$\mathcal{I} \models T_2 \rightarrow \mathcal{I} \models T_1$$

**Theorem 2.2.1.** It is undecidable to determine if a first order logic formula is a tautology.

**Equivalence** A sentence  $T_1$  is equivalent to  $T_2$  if  $T_1 \models T_2$  and  $T_2 \models T_1$ .

Equivalence

**Theorem 2.2.2.** The following statements are equivalent:

1.  $F_1, \dots, F_n \models G$ .
2.  $(\bigwedge_{i=1}^n F_i) \rightarrow G$  is valid.
3.  $(\bigwedge_{i=1}^n F_i) \wedge \neg G$  is unsatisfiable.

## 2.3 Substitution

**Substitution** A substitution  $\sigma : \mathcal{V} \rightarrow \mathcal{T}$  is a mapping from variables to terms. It is written as  $\{X_1 \mapsto t_1, \dots, X_n \mapsto t_n\}$ .

Substitution

The application of a substitution is the following:

- $p(t_1, \dots, t_n)\sigma = p(t_1\sigma, \dots, t_n\sigma)$
- $f(t_1, \dots, t_n)\sigma = fp(t_1\sigma, \dots, t_n\sigma)$
- $\perp\sigma = \perp$  and  $\top\sigma = \top$
- $(\neg F)\sigma = (\neg F\sigma)$
- $(F_1 \star F_2)\sigma = (F_1\sigma \star F_2\sigma)$  for  $\star \in \{\wedge, \vee, \rightarrow\}$



- $(\forall X.F)\sigma = \forall X'(F\sigma[X \mapsto X'])$  where  $X'$  is a fresh variable (i.e. does not appear in  $F$ ).
- $(\exists X.F)\sigma = \exists X'(F\sigma[X \mapsto X'])$  where  $X'$  is a fresh variable.

**Unifier** A substitution  $\sigma$  is a unifier for  $e_1, \dots, e_n$  if  $e_1\sigma = \dots = e_n\sigma$ .

Unifier

**Most general unifier** A unifier  $\sigma$  is the most general unifier (MGU) for  $\bar{e} = e_1, \dots, e_n$  if every unifier  $\tau$  for  $\bar{e}$  is an instance of  $\sigma$  ( $\tau = \sigma\rho$  for some substitution  $\rho$ ). In other words,  $\sigma$  is the smallest substitution to unify  $\bar{e}$ .

Most general unifier

## 3 Logic programming

### 3.1 Syntax

A logic program has the following components (defined using BNF):

**Atom**  $A := p(t_1, \dots, t_n)$  for  $n \geq 0$

Atom

**Goal**  $G := \top \mid \perp \mid A \mid G_1 \wedge G_2$

Goal

**Horn clause** A clause with at most one positive literal.

Horn clause

$$K := A \leftarrow G$$

In other words,  $A$  and all the literals in  $G$  are positive as  $A \leftarrow G = A \vee \neg G$ .

**Program**  $P := K_1 \dots K_m$  for  $m \geq 0$

Program

### 3.2 Semantics

#### 3.2.1 State transition system

**State** Pair  $\langle G, \theta \rangle$  where  $G$  is a goal and  $\theta$  is a substitution.

State

**Initial state**  $\langle G, \varepsilon \rangle$

**Successful final state**  $\langle \top, \theta \rangle$

**Failed final state**  $\langle \perp, \varepsilon \rangle$

**Derivation** A sequence of states. A derivation can be:

Derivation

**Successful** If the final state is successful.

**Failed** If the final state is failed.

**Infinite** If there is an infinite sequence of states.

Given a derivation, a goal  $G$  can be:

**Successful** There is a successful derivation starting from  $\langle G, \varepsilon \rangle$ .

**Finitely failed** All the derivations starting from  $\langle G, \varepsilon \rangle$  are failed.

**Computed answer substitution** Given a goal  $G$  and a program  $P$ , if there exists a successful derivation  $\langle G, \varepsilon \rangle \mapsto^* \langle \top, \theta \rangle$ , then the substitution  $\theta$  is the computed answer substitution of  $G$ .

Computed answer substitution

**Transition** Starting from the state  $\langle A \wedge G, \theta \rangle$  of a program  $P$ , a transition on the atom  $A$  can result in:

Transition

**Unfold** If there exists a clause  $(B \leftarrow H)$  in  $P$  and a (most general) unifier  $\beta$  for  $A\theta$  and  $B$ , then we have a transition:  $\langle A \wedge G, \theta \rangle \mapsto \langle H \wedge G, \theta\beta \rangle$ .

In other words, we want to prove that  $A\theta$  holds. To do this, we search for a clause that has as conclusion  $A\theta$  and add its premise to the things to prove. If a unification is needed to match  $A\theta$ , we add it to the substitutions of the state.

**Failure** If there are no clauses  $(B \leftarrow H)$  in  $P$  with a unifier for  $A\theta$  and  $B$ , then we have a transition:  $\langle A \wedge G, \theta \rangle \mapsto \langle \perp, \varepsilon \rangle$ .

**Non-determinism** A transition has two types of non-determinism:

**Don't-care** Any atom in  $(A \wedge G)$  can be chosen to determine the next state. Don't-care  
This affects the length of the derivation (infinite in the worst case).

**Don't-know** Any clause  $(B \rightarrow H)$  in  $P$  with an unifier for  $A\theta$  and  $B$  can be chosen. This determines the output of the derivation. Don't-know

**Selective linear definite resolution** Approach to avoid non-determinism when constructing a derivation. SLD resolution

**Selection rule** Method to select the atom in the goal to unfold (eliminates don't-care non-determinism). Selection rule

**SLD tree** Search tree constructed using all the possible clauses according to a selection rule and visited following a search strategy (eliminates don't know non-determinism). SLD tree

**Theorem 3.2.1** (Soundness). Given a program  $P$ , a goal  $G$  and a substitution  $\theta$ , if  $\theta$  is a computed answer substitution, then  $P \models G\theta$ .

**Theorem 3.2.2** (Completeness). Given a program  $P$ , a goal  $G$  and a substitution  $\theta$ , if  $P \models G\theta$ , then it exists a computed answer substitution  $\sigma$  such that  $G\theta = G\sigma\beta$ .

**Theorem 3.2.3.** If a computed answer substitution can be obtained using a selection rule  $r$ , it can be obtained also using a different selection rule  $r'$ .

**Prolog SLD** Prolog SLD

**Selection rule** Select the leftmost atom.

**Tree search strategy** Search following the order of definition of the clauses.

This results in a left-to-right, depth-first search of the SLD tree. Note that this may end up in a loop.

## 4 Prolog

It may be useful to first have a look at the "Logic programming" section of *Languages and Algorithms for AI (module 2)*.

### 4.1 Syntax

**Term** Following the first-order logic definition, a term can be a:

Term

- Constant (`lowerCase`).
- Variable (`UpperCase`).
- Function symbol (`f(t1, ..., tn)` with `t1, ..., tn` terms).

**Atomic formula** An atomic formula has form:

Atomic formula

$$p(t_1, \dots, t_n)$$

where `p` is a predicate symbol and `t1, ..., tn` are terms.

Note: there are no syntactic distinctions between constants, functions and predicates.

**Clause** A Prolog program is a set of horn clauses:

Horn clause

**Fact** `A`.

**Rule** `A :- B1, ..., Bn`. (`A` is the head and `B1, ..., Bn` the body)

**Goal** `:- B1, ..., Bn`.

where:

- `A, B1, ..., Bn` are atomic formulas.
- `,` represents the conjunction ( $\wedge$ ).
- `:-` represents the logical implication ( $\Leftarrow$ ).

**Quantification**

Quantification

**Facts** Variables appearing in a fact are quantified universally.

$$A(X) . \equiv \forall X : A(X)$$

**Rules** Variables appearing the the body only are quantified existentially. Variables appearing in both the head and the body are quantified universally.

$$A(X) :- B(X, Y) . \equiv \forall X, \exists Y : A(X) \Leftarrow B(X, Y)$$

**Goals** Variables are quantified existentially.

$$:- B(Y) . \equiv \exists Y : B(Y)$$

## 4.2 Semantics

**Execution of a program** A computation in Prolog attempts to prove the goal. Given a program  $P$  and a goal  $:- p(t_1, \dots, t_n)$ , the objective is to find a substitution  $\sigma$  such that:

$$P \models [p(t_1, \dots, t_n)]\sigma$$

In practice, it uses two stacks:

**Execution stack** Contains the predicates the interpreter is trying to prove.

**Backtracking stack** Contains the choice points (clauses) the interpreter can try.

**SLD resolution** Prolog uses SLD resolution with the following choices:

SLD

**Left-most** Always proves the left-most literal first.

**Depth-first** Applies the predicates following the order of definition.

Note that the depth-first approach can be efficiently implemented (tail recursion) but the termination of a Prolog program on a provable goal is not guaranteed as it may loop depending on the ordering of the clauses.

**Disjunction operator** The operator `;` can be seen as a disjunction and makes the Prolog interpreter explore the remaining SLD tree looking for alternative solutions.

## 4.3 Arithmetic operators

In Prolog:

Arithmetic operators

- Integers and floating points are built-in atoms.
- Math operators are built-in function symbols.

Therefore, mathematical expressions are terms.

**is predicate** The predicate `is` is used to evaluate and unify expressions:

$$T \text{ is Expr}$$

where  $T$  is a numerical atom or a variable and `Expr` is an expression without free variables. After evaluation, the result of `Expr` is unified with  $T$ .

**Example.**

```
?- X is 2+3.
    yes X=5
```

Note: a term representing an expression is evaluated only with the predicate `is` (otherwise it remains as is).

**Relational operators** Relational operators (`>`, `<`, `>=`, `<=`, `==`, `=/=`) are built-in.

## 4.4 Lists

A list is defined recursively as:

Lists

**Empty list** `[]`

**List constructor** `.(T, L)` where T is a term and L is a list.

Note that a list always ends with an empty list.

As the formal definition is impractical, some syntactic sugar has been defined:

**List definition** `[t1, ..., tn]` can be used to define a list.

**Head and tail** `[H | T]` where H is the head (term) and T the tail (list) can be useful for recursive calls.

## 4.5 Cut

The cut operator `(!)` allows to control the exploration of the SLD tree.

Cut

A cut in a clause:

`p :- q1, ..., qi, !, qj, ..., qn.`

makes the interpreter consider only the first choice points for `q1, ..., qi`, dropping all the other possibilities. Therefore, if `qj, ..., qn` fails, there won't be backtracking and `p` fails.

**Example.**

```
p(X) :- q(X), r(X).
q(1).
q(2).
r(2).
```

```
?- p(X).
yes X=2
```

```
p(X) :- q(X), !, r(X).
q(1).
q(2).
r(2).
```

```
?- p(X).
no
```

In the second case, the cut drops the choice point `q(2)` and only considers `q(1)`.

**Mutual exclusion** A cut can be useful to achieve mutual exclusion. In other words, to represent a conditional branching:

`if a(X) then b else c`

a cut can be used as follows:

```
p(X) :- a(X), !, b.
p(X) :- c.
```

If `a(X)` succeeds, other choice points for `p` will be dropped and only `b` will be evaluated. If `a(X)` fails, the second clause will be considered, therefore evaluating `c`.

## 4.6 Negation

**Closed-world assumption** Only what is stated in a program  $P$  is true, everything else is false:

Closed-world assumption

$$\text{CWA}(P) = P \cup \{\neg A \mid A \text{ is a ground atomic formula and } P \not\models A\}$$

**Non-monotonic inference rule** Adding new axioms to the program may change the set of valid theorems.

As first-order logic is undecidable, closed-world assumption cannot be directly applied in practice.

**Negation as failure** A negated atom  $\neg A$  is considered true iff  $A$  fails in finite time:

Negation as failure

$$\text{NF}(P) = P \cup \{\neg A \mid A \in \text{FF}(P)\}$$

where  $\text{FF}(P) = \{B \mid P \not\models B \text{ in finite time}\}$  is the set of atoms for which the proof fails in finite time. Note that not all atoms  $B$  such that  $P \not\models B$  are in  $\text{FF}(P)$ .

**SLDNF** SLD resolution with NF to solve negative atoms.

SLDNF

Given a goal of literals  $:- L_1, \dots, L_m$ , SLDNF does the following:

1. Select a positive or ground negative literal  $L_i$ :
  - If  $L_i$  is positive, apply the normal SLD resolution.
  - If  $L_i = \neg A$ , prove that  $A$  fails in finite time. If it succeeds,  $L_i$  fails.
2. Solve the goal  $:- L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_m$ .

**Theorem 4.6.1.** If only positive or ground negative literal are selected during resolution, SLDNF is correct and complete.

**Prolog SLDNF** Prolog uses an incorrect implementation of SLDNF where the selection rule always chooses the left-most literal. This potentially causes incorrect deductions.

*Proof.* When proving  $:- \text{capital}(X)$ , the intended meaning is:

$$\exists X : \neg \text{capital}(X)$$

In SLDNF, to prove  $:- \text{capital}(X)$ , the algorithm proves  $:- \text{capital}(X)$ , which results in:

$$\exists X : \text{capital}(X)$$

and then negates the result, which corresponds to:

$$\neg(\exists X : \text{capital}(X)) \iff \forall X : (\neg \text{capital}(X))$$

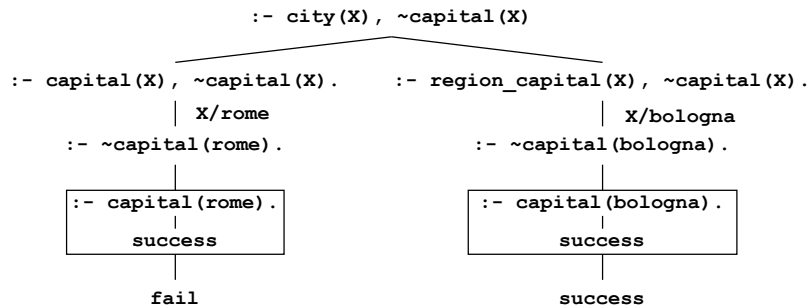
□

**Example** (Correct SLDNF resolution). Given the program:

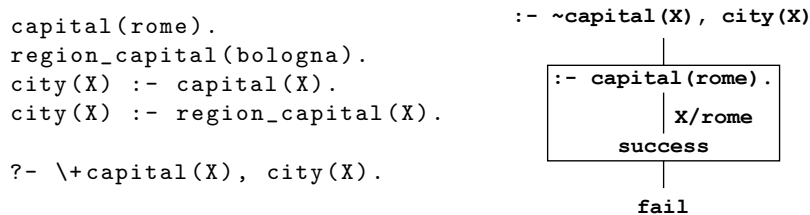
```
capital(rome).
region_capital(bologna).
city(X) :- capital(X).
city(X) :- region_capital(X).

?- city(X), \+capital(X).
```

its resolution succeeds with  $X=bologna$  as  $\backslash+capital(X)$  is ground by the unification of  $city(X)$ .



**Example** (Incorrect SLDNF resolution). Given the program:



its resolution fails as  $\backslash+capital(X)$  is a free variable and the proof of  $capital(X)$  is ground with  $X=rome$  and succeeds, therefore failing  $\backslash+capital(X)$ . Note that  $bologna$  is not tried as it does not appear in the axioms of  $capital$ .

## 4.7 Meta predicates

**call/1** Given a term  $T$ ,  $call(T)$  considers  $T$  as a predicate and evaluates it. At the time of evaluation,  $T$  must be a non-numeric term. call/1

**Example.**

```

p(X) :- call(X).
q(a).

?- p(q(Y)).
yes Y=a
  
```

**fail/0** The evaluation of **fail** always fails, forcing the interpreter to backtrack. fail/0

**Example** (Implementation of negation as failure).

```

not(P) :- call(P), !, fail.
not(P).
  
```

Note that the cut followed by **fail** (**!, fail**) is useful to force a global failure.

**bagof/3 and setof/3**

**bagof/3** The predicate **bagof**( $X, P, L$ ) unifies  $L$  with a list of the instances of  $X$  that satisfy  $P$ . Fails if none exists. bagof/3

**setof/3** The predicate **setof**( $X, P, S$ ) unifies  $S$  with a set of the instances of  $X$  that satisfy  $P$ . Fails if none exists. setof/3

In practice, for computational reasons, a list (with repetitions) might be computed.



**Example.**

```
p(1).
p(2).
p(1).

?- setof(X, p(X), S).
   yes S=[1, 2] X=X

?- bagof(X, p(X), S).
   yes S=[1, 2, 1] X=X
```

**Quantification** When solving a goal, the interpreter unifies free variables with a value. This may cause unwanted behaviors when using `bagof` or `setof`. The `X^` tells the interpreter to not (permanently) bind the variable `X`.

**Example.**

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).

?- setof(X, father(X, Y), S).
   yes X=X Y=giuseppe S=[giovanni];
      X=X Y=mario     S=[giovanni];
      X=X Y=paola     S=[mario]
```

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).

?- setof(X, Y^father(X, Y), S).
   yes S=[giovanni, mario] X=X Y=Y
```

**findall/3** The predicate `findall(X, P, S)` unifies `S` with a list of the instances of `X` that satisfy `P`. If none exists, `S` is unified with an empty list. Variables in `P` that do not appear in `X` are not bound (same as the `Y^` operator).

**Example.**

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).

?- findall(X, father(X, Y), S).
   yes S=[giovanni, mario] X=X Y=Y
```

**var/1** The predicate `var(T)` is true if `T` is a variable. var/1

**nonvar/1** The predicate `nonvar(T)` is true if `T` is not a free variable. nonvar/1

**number/1** The predicate `number(T)` is true if `T` is a number. number/1

**ground/1** The predicate `ground(T)` is true if `T` does not have free variables. ground/1

**=../2** The operator `T =.. L` unifies `L` with a list where its head is the head of `T` and the tail contains the remaining arguments of `T` (i.e. puts all the components of a predicate into a list). Only one between `T` and `L` may be a variable. =../2

**Example.**

```
?- foo(hello, X) =.. List.
   List = [foo, hello, X]

?- Term =.. [baz, foo(1)].
   Term = baz(foo(1))
```

**clause/2** The predicate `clause(Head, Body)` is true if it can unify `Head` and `Body` with an existing clause. `Head` must be initialized to a non-numeric term. `Body` can be a variable or a term. clause/2

**Example.**

```
p(1).
q(X, a) :- p(X), r(a).
q(2, Y) :- d(Y).

?- clause(p(1), B).
   yes B=true

?- clause(p(X), true).
   yes X=1

?- clause(q(X, Y), B).
   yes X=_1 Y=a B=p(_1), r(a);
      X=2 Y=_2 B=d(_2)
```

**assert/1** The predicate `assert(T)` adds `T` in an unspecified position of the clauses database of Prolog. In other words, it allows to dynamically add clauses. assert/1

**asserta/1** As `assert(T)`, with insertion at the beginning of the database. asserta/1

**assertz/1** As `assert(T)`, with insertion at the end of the database. assertz/1

Note that `:- assert((p(X)))` quantifies `X` existentially as it is a query. If it is not ground and added to the database as is, it becomes a clause and therefore quantified universally:  $\forall X : p(X)$ .

**Example** (Lemma generation).

```
fib(0, 0) :- !.
fib(1, 1) :- !.
fib(N, F) :- N1 is N-1, fib(N1, F1),
             N2 is N-2, fib(N2, F2),
             F is F1+F2,
             generate_lemma(fib(N, F)).

generate_lemma(T) :- clause(T, true), !.
generate_lemma(T) :- assert(T).
```

**generate\_lemma/1** allows to add to the clauses database all the intermediate steps to compute the Fibonacci sequence (similar concept to dynamic programming).

**retract/1** The predicate `retract(T)` removes from the database the first clause that unifies with `T`. retract/1

**abolish/2** The predicate `abolish(T, n)` removes from the database all the occurrences of `T` with arity `n`. abolish/2

## 4.8 Meta-interpreters

**Meta-interpreter** Interpreter for a language  $L_1$  written in another language  $L_2$ . Meta-interpreter

**Prolog vanilla meta-interpreter** The Prolog vanilla meta-interpreter is defined as follows: Vanilla  
meta-interpreter

```
solve(true) :- !.  
solve( (A, B) ) :- !, solve(A), solve(B).  
solve(A) :- clause(A, B), solve(B).
```

In other words, the clauses state the following:

1. A tautology is a success.
2. To prove a conjunction, we have to prove both atoms.
3. To prove an atom A, we look for a clause  $A :- B$  that has A as conclusion and prove its premise B.

# 5 Constraint programming

## Class of problems

**Constraint satisfaction problem (CSP)** Defined by:

- A finite set of variables  $X_1, \dots, X_n$ .
- A domain for each variable  $D(X_1), \dots, D(X_n)$ .
- A set of constraints  $\{C_1, \dots, C_m\}$

A solution is an assignment to all the variables while satisfying the constraints.

**Constraint optimization problem (COP)** Extension of a constraint satisfaction problem with an objective function with domain  $D$ :

$$f : D(X_1) \times \dots \times D(X_n) \rightarrow D$$

A solution is a CSP solution that optimizes  $f$ .

Constraint  
satisfaction problem

Constraint  
optimization  
problem

## Class of languages

**Constraint logic programming (CLP)** Add constraints and solvers to logic programming. Generally more efficient than plain logic programming.

**Imperative languages** Add constraints and solvers to imperative languages.

Constraint logic  
programming

Imperative  
languages

## 5.1 Constraint logic programming (CLP)

### 5.1.1 Syntax

**Atom**  $A := p(t_1, \dots, t_n)$ , for  $n \geq 0$ .  $p$  is a predicate.

**Constraint**  $C := c(t_1, \dots, t_n) \mid C_1 \wedge C_2$ , for  $n \geq 0$ .  $c$  is an atomic constraint.

**Goal**  $G := \top \mid \perp \mid A \mid C \mid G_1 \wedge G_2$

**Constraint logic clause**  $K := A \leftarrow G$

**Constraint logic program**  $P := K_1 \dots K_m$ , for  $m \geq 0$

Atom

Constraint

Goal

Constraint logic  
clause

Constraint logic  
program

### 5.1.2 Semantics

#### Transition system

**State** Pair  $\langle G, C \rangle$  where  $G$  is a goal and  $C$  is a constraint.

**Initial state**  $\langle G, \top \rangle$

**Successful final state**  $\langle \top, C \rangle$  with  $C \neq \perp$

**Failed final state**  $\langle G, \perp \rangle$

**Transition** Starting from the state  $\langle A \wedge G, C \rangle$  of a program  $P$ , a transition on the atom  $A$  can result in:

Transition system

**Unfold** If there exists a clause  $(B \leftarrow H)$  in  $P$  and an assignment  $(B \doteq A)$  such that  $((B \doteq A) \wedge C)$  is still valid, then we have a transition  $\langle A \wedge G, C \rangle \mapsto \langle H \wedge G, (B \doteq A) \wedge C \rangle$ . Unfold

In other words, we want to develop an atom  $A$  and the current constraints are denoted as  $C$ . We look for a clause whose head equals  $A$ , applying an assignment if needed. If this is possible, we transit from solving  $A$  to solving the body of the clause and add the assignment to the set of active constraints.

**Failure** If there are no clauses  $(B \leftarrow H)$  with a valid assignment  $((B \doteq A) \wedge C)$ , then we have a transition  $\langle A \wedge G, C \rangle \mapsto \langle \perp, \perp \rangle$ . Failure

Moreover, starting from the state  $\langle C \wedge G, D_1 \rangle$  of a program  $P$ , a transition on the constraint  $C$  can result in:

**Solve** If  $(C \wedge D_1) \iff D_2$  holds, then we have a transition  $\langle C \wedge G, D_1 \rangle \mapsto \langle G, D_2 \rangle$ . Solve

In other words, we want to develop a constraint  $C$  and the current constraints are denoted as  $D_1$ . If  $(C \wedge D_1)$  is valid, we call it  $D_2$  and continue solving the rest of the goal constrained to  $D_2$ .

**Non-determinism** As in logic programming, there is don't-care and don't-know non-determinism. A SLD search tree is also used.

## Derivation strategies

<b>Generate-and-test</b>	Strategy adopted by logic programs. Every possible assignment to the variables are generated and tested.	Generate-and-test
--------------------------	--	-------------------

The workflow is the following:

1. Determine domain.
2. Make an assignment to each variable.
3. Test the constraints.

<b>Constrain-and-generate</b>	Strategy adopted by constraint logic programs. Exploit facts to reduce the search space.	Constrain-and-generate
-------------------------------	--	------------------------

The workflow is the following:

1. Determine domain.
2. Restrict the domain following the constraints.
3. Make an assignment to each variable.

## 5.2 MiniZinc

Declarative language for constraint programming.

**Built-in types** bool, int, float, string Built-in types

Logical operators  $\vee \cdot \wedge \cdot \rightarrow \cdot !$  Logical operators

<b>Parameter</b>	User-inputted value passed to the solver before execution.	Parameter
------------------	--	-----------

<domain>: <name>

**Example.** `int: size;`

**Variable** Value computed by the solver.

Variable

`var <domain>: <name>`

**Example.** `var bool: flag;`

**Set** For defining ranges.

Set

`set of <domain>: <name>`

**Example.** `set of int: top10 = 1..10;`

**Array** Array of parameters or variables.

Array

`array[<index range>] of <domain>: <name>`

**Example.** `array[1..5] of var int: vars;`

**Aggregation functions** `sum, product, min, max.`

Aggregation  
functions

**Forall**

`forall(<iterators> in <domain>)(<conditions>)`  
`forall(<iterators> in <domain> where <conditions>)(<conditions>)`

**Example.** `forall(i, j in 1..3 where i < j)(arr[i] != arr[j]);`

**Exists**

`exists(<iterators> in <domain>)(<conditions>)`  
`exists(<iterators> in <domain> where <conditions>)(<conditions>)`

**Constraints**

Constraints

`constraint <expression>`

Multiple constraints are seen as conjunctions.

**Example.** `constraint X >= 5 /\ X != 10;`

**Global constraints** `all_different(...), all_equal(...)`

**Solver**

Solver

**Satisfiability problem**

`solve satisfy;`

**Optimization problem**

`solve minimize <variable>;`

<end of course>