Languages and Algorithms for Artificial Intelligence

Module 2: Logic and Logic Programming

Alessio Arcara Alessia Crimaldi

Contents

1 Logic

Truth. The notion of truth is always defined with respect to a world. The problem is when the world changes or when we adopt new world; in such cases, the truth also changes. Therefore, there isn't something that is universally true, and it is better to replace it with the concept of logical consequence, which doesn't change with the world.

Definition 1: Logical Consequence

For a set of sentences $\Gamma = F_1, \ldots, F_n$, we say that F is a logical consequence of Γ in various possible worlds, if it is always true that: whenever all sentences in Γ are true, then F is true.

Definition 2: Logical Equivalence

if F is logical consequence of G and G is logical consequence of F, then they are equivalent.

Now, the notion of truth becomes contextual rather than universal, and the world is characterized by the sentences that describe it.

1.1 Propositional logic

Propositional logic is the logic which deals only with propositions.

Definition 3: Proposition

A proposition P_i is a statement about some specific fact which can be true or false. P_i is an atom or atomic formula.

Example

- Today it is raining
- I take the umbrella
- There is the sun

We can build larger sentences from smaller ones by using **connectives**. While natural language provides a rich source of connectives, we opt for connectives that do not introduce confusion to build an unambiguous artificial language. For the same reason, we do not use natural language because it is ambiguous by its nature. Consequently, our goal is to build a precise artificial language that is not ambiguous.

Example

- John drove on and hit a pedestrian.
- John hit a pedestrian and drove on.
- If I open the window then 1+3=4

To do so, the logic, like any formal system, needs to define both **syntax** and **semantics**. Syntax specify the rules which tell us how the well formed sentences are constructed, while semantics specify the rules which tell us the meaning of the well formed sentences.

Definition 4: Alphabet

The language of propositional logic has an alphabet consisting of

1. proposition symbols: $p_0, p_1, \ldots,$
2. connectives: $\wedge, \vee, \rightarrow, \neg, \leftrightarrow, \bot,$
3. auxiliary symbols: $(,)$.

The connectives carry traditional names:

$\wedge$	- and	- conjunction
$\vee$	- or	- disjunction
$\rightarrow$	- if ..., then ...	- implication
$\neg$	- not	- negation
$\leftrightarrow$	- iff	- equivalence, bi-implication
$\bot$	- falsity	- falsum, absurdum

The proposition symbols and $\bot$ stand for indecomposable propositions, which we call **atoms**, or **atomic propositions**.

Well-formed formula is defined recursively:

$$\begin{array}{lll} \text{formula} & ::= & \text{Atomic Proposition} \\ & | & \neg \text{ formula} \\ & | & \text{formula} \wedge \text{formula} \\ & | & \text{formula} \vee \text{formula} \\ & | & \text{formula} \rightarrow \text{formula} \\ & | & \text{formula} \leftrightarrow \text{formula} \\ & | & (\text{formula}) \end{array}$$

If we write the proposition "Today it is raining" is it true or false? Of course, as said in previous section, it depends on the world we are considering, so we need the notion of **interpretation**.

Definition 5: Interpretation

Given a formula G, let $\{A_1, \ldots, A_n\}$ be a set of atoms which occur in the formula, an interpretation I of G is an assignment of truth values to $\{A_1, \ldots, A_n\}$.

The semantics of a formula are defined by the truth values of its atoms and their corresponding connectives. We can represent each possible combination of truth values in a structure format known as a **truth table**.

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

It's important to note that implication can be a bit tricky: if p_1 is true, then p_2 must also be true; however, if p_1 is false, any value is okay for p_2.

Exam

Write the truth table for the formula $G = (P \vee Q) \wedge \neg(P \wedge Q)$:

P	Q	$P \vee Q$	$P \wedge Q$	$\neg(P \wedge Q)$	G
T	T	T	T	F	F
T	F	T	F	T	T
F	T	T	F	T	T
F	F	F	F	T	F

Definition 6: Model

We say that an interpretation INT is a model of a formula F, written

$$INT \models F$$

if F is true when the truth value of propositional symbols is defined according to INT.

Definition 7: Validity

A formula F is valid iff it is true in all its interpretation. If F is valid we can write $\models F$.

Example

$A \vee \neg A$ is a valid formula.

Definition 8: Satisfiability

A formula F is satisfiable iff there exists an assignment of truth values that can make the formula true. If F is satisfiable we can write $I \models F$.

Example

$A \wedge \neg A$ is an unsatisfiable formula.

Definition 9: Decidability

Propositional logic is decidable: there is a terminating procedure to check whether a formula is valid.
The algorithm is simple: write down the complete truth table for the formula and check if the last column contains only T. Such algorithm has a computation cost of 2^n, where n is the number of interpretations.

Definition 10: 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 every interpretation of F and G.

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

Definition 11: Literal

A literal is an atom or the negation of an atom.

Definition 12: Negation Normal Form (NNF)

A formula is in Negation Normal Form iif negations appears only in front of atoms.

Definition 13: Conjunctive Normal Form (CNF)

A formula F is in Conjunctive Normal Form iif it is in NNF and it has the form $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ where each F_i is a disjunction of literals.

Example

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

Definition 14: Disjunctive Normal Form (CNF)

A formula F is in Disjunctive Normal Form iif it is in NNF and it has the form $F_1 \vee F_2 \vee \ldots \vee F_n$ where each F_i is a conjunction of literals.

Example

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

Any formula can be transformed into a normal form by using the equivalence rules. Normal form is easier to evaluate for a machine.

Exam

Example of transformation in CNF

$$
\begin{aligned}
(P \rightarrow Q) \wedge (R \vee (B \wedge A)) &\equiv \\
&\equiv (\neg P \vee Q) \wedge (R \vee (B \wedge A)) \\
&\equiv (\neg P \vee Q) \wedge ((R \vee B) \wedge (R \vee A)) \\
&\equiv (\neg P \vee Q) \wedge (R \vee B) \wedge (R \vee A)
\end{aligned}
$$

Basically, we begin by applying implication elimination, followed by the pushing of negations within the formula. After that we apply the distribution of connectives.

Definition 15: Logical Consequence

Given a set of formulas $\{F_1, \ldots, F_n\}$ and a formula G, G is said be a logical consequence of $F_1, \ldots, F_n$, denoted as $F_1 \wedge \ldots \wedge F_n \models G$, iff for any interpretation I in which $F_1 \wedge \ldots \wedge F_n$ is true G is also true.

Instead of constructing the truth table, the following theorems show that we can prove logical consequence by proving validity of a formula or proving a given formula is never satisfiable.

Theorem 1: Deduction

Given a set of formulas $\{F_1, \ldots, F_n\}$ and a formula G, $F_1 \wedge \ldots \wedge F_n \models G$ iif $\models (F_1 \wedge \ldots F_n) \rightarrow G$ (is valid).

Theorem 2: Refutation

Given a set of formulas $\{F_1, \ldots, F_n\}$ and a formula G, $(F_1 \wedge \ldots \wedge F_n) \models G$ iif $F_1 \wedge \ldots \wedge F_n \neg G$ is never satisfiable.

Example

To determine if an argument is valid in propositional logic, the initial step is to identify the basic propositions from natural language and disregard anything irrelevant (e.g. quantifiers).
All the dated letters in this room are written on blue paper,
This is decomposed as:

- P: the letter is dated;
- Q: the letter is written on blue paper;
- $P \to Q$.

None of them are in black ink, except those that are written in third person.
This is decomposed as:

- S: letters is written in the third person.
- R: the letter is written in black ink.
- $\neg S \to \neg R$

Exam

If John drinks beer, he is at least 18 years old. John does not drink beer. Therefore, John is not yet 18 years old.

- D: John drinks beer
- Y: John is at least 18 year old
- $\neg D$
- $D \to Y$

We want to prove that $D \to Y \wedge \neg D \models \neg Y$, assuming all the sentences are true. We can do this using three different methods:

- truth table

D	Y	$\neg D$	$D \to Y$	$\neg D \wedge (D \to Y)$	$\neg Y$
T	T	F	T	F	F
T	F	F	F	F	T
F	T	T	T	T	F
F	T	T	T	T	T

- deduction: $\models (D \to Y \wedge \neg D) \to \neg Y$
- refutation: $D \to Y \wedge \neg D \wedge Y$ is never satisfiable

1.2 Natural deduction

We have seen a way to prove logical consequences based on semantics, using truth tables. However, we can also prove logical consequences through syntax, which is much more efficient. This involves performing syntactic manipulations of the symbols through a sequence of steps. Each step represents a derivation of a conclusion from the given premises. That can be expressed as:

$$\frac{\text{premises}}{\text{conclusion}}$$

The derivation perform either elimination or introduction of connectives, using the following derivation rules:

- **Conjuction introduction**: if A is true and B is true we may conclude $A \wedge B$ is true.

$$\frac{A \quad B}{A \wedge B} \wedge I$$

- **Conjunction elimination**: if $A \wedge B$ is true we may conclude A is true (or B is true).

$$\frac{A \wedge B}{A} \wedge E \quad \text{or} \quad \frac{A \wedge B}{B} \wedge E$$

- **Implication introduction**: if B follows from A, then we may conclude $A \to B$.

$$\begin{array}{c} [A] \\ \vdots \\ \dfrac{B}{A \to B} \to I \end{array}$$

Note that implication is only valid along the specific path of the proof tree. Once the implication is derived, the initial hypothesis becomes redundant and can be discarded (denoted as []), as it is inherently contained within the implication.

- **Implication elimination**: if A true and we know that B follows from A, then we have also B is true.

$$\frac{A \quad A \to B}{B} \to E$$

- **Ex falso sequitur quodlibet**: if we have a false premise, we can derive any possible formula.

$$\frac{\bot}{A} \bot$$

- **Reduction Ad Absurdum**: if one derives a contradiction from the hypothesis $\neg A$, then one has a derivation of A (without the hypothesis $\neg A$).

$$\begin{array}{c} [\neg A] \\ \vdots \\ \dfrac{\bot}{A} \text{ RAA} \end{array}$$

In that process, a deduction tree is built. Each internal node represents a derivation step, with the root corresponding to the formula, and the hypotheses are the leaves of the tree.
We restrict our language to the connectives $\wedge, \to$ and $\bot$. Negation can be express as $A \to \bot$.

Example

$$\frac{\dfrac{A \wedge B}{A}\,(\wedge E) \qquad \dfrac{C \wedge D}{D}\,(\wedge E)}{A \wedge D}\,(\wedge I)$$

Example

Demonstrate commutativity $A \wedge B \to B \wedge A$ using only natural deduction

$$\frac{\dfrac{\dfrac{[A \wedge B]^1}{B} \wedge E \qquad \dfrac{[A \wedge B]^1}{A} \wedge E}{B \wedge A} \wedge I}{A \wedge B \to B \wedge A} \to I_1$$

At last, it has been used the implication introduction and it has been discarded the hypothesis to prove that $B \to A$ is always true.

Example

Provide the proof for $A \to \neg\neg A \equiv A \to \neg(A \to \bot) \equiv A \to ((A \to \bot) \to \bot)$.

$$\frac{\dfrac{\dfrac{[A]^2 \qquad [A \to \bot]^1}{\bot} \to E}{((A \to \bot) \to \bot)} \to I_1}{(A \to ((A \to \bot) \to \bot)} \to I_2$$

The derivation process serves the purpose to proving validity of a given formula. However, when there is a suspect that a formula might be false, we need to identify an assignment of truth values that makes the formula false.

Definition 16

We write $\Gamma \vdash \psi$ if there exists a deduction tree with the conclusion ψ as the root and the premises Γ, which are not discarded, as the leaves.
When $\Gamma = \emptyset$, so all premises are discarded, we say that ψ is a **theorem**.

Theorem 3: Completeness

$$\Gamma \vdash \psi \Leftrightarrow \Gamma \models \psi$$

This theorem tells us that what we derive is true, and all that is true can be derived.

Exam

$A \to (B \to C) \to (A \wedge B \to C)$

To prove the validity of the given formula, we can use only the natural deduction method. The goal is to demonstrate the truth by discarding all the hypotheses in the deduction tree.

Before proceeding with the proof, it's important to evaluate whether the given formula appears more reasonably true or false. Otherwise, we could spend half an hour proving the wrong thing.

The formula provides us a strategy for tackling the problem. One approach can be to proceed backward to validate the formula:

1. $\dfrac{A \wedge B \to C}{A \to (B \to C) \to (A \wedge B \to C)} \to I$
2. $\dfrac{C}{A \wedge B \to C} \to I$
3. $\dfrac{B \quad B \to C}{C} \to E$
   (a) $\dfrac{A \wedge B}{B} \wedge E$
   (b) $\dfrac{A \quad A \to (B \to C)}{B \to C} \to E$
      i. $\dfrac{A \wedge B}{A} \wedge E$

Finally, I can discard the hypothesis leaves: (a), i. with the introduction of implication 2 and (b) with the introduction of implication 1. Therefore, this holds without any hypothesis, thus proving it.

1.3 First-order logic

First-order logic can be seen as an extension of propositional logic. In propositional logic the atomic formulas are propositional variables that are either true or false. In first-order logic the atomic formulas are *predicates* that assert a relationship among certain elements. Another significant new concept in first-order logic is *quantification*: the ability to assert that a certain property holds *for all elements* or that it holds *for some element*.

Syntax

Definition 17: Alphabet

The alphabet consists of the following symbols:

- $\mathcal{P}$: predicate symbols: $p, q, r, \ldots$
- $\mathcal{F}$: function symbols: $a, b, c, \ldots$
- $\mathcal{V}$: variables: $X, Y, Z, \ldots$
- logic symbols:
  - truth symbols: $\bot$ $\top$
  - logical connectives: $\neg$ $\wedge$ $\vee$ $\rightarrow$
  - quantifiers: $\forall$ $\exists$
  - syntactic symbols: ( ) ,

A function takes $n \geq 0$ arguments and returns a *value*. A predicate is a relation that takes $n \geq 0$ arguments, and is either true or false. If we have a variable, it is the set of the values of the variable where the relation becomes true.

Definition 18: Term

A term is either a variable from $\mathcal{V}$ or a function symbol applied to $n \geq 0$ arguments, where each of these n arguments is also a term.

A **constant** is a function symbol with no arguments.

Example

$(a/0,\ f/1)$:

- X
- a
- $f(X)$

An **atomic formula** is a predicate symbol p with terms as its arguments. It is important to note that in first-order logic, predicates cannot be used as arguments for other predicates. Complex formulas can be constructed through the use of connectives.

Example

$\mathcal{P} = \{\text{mortal}/1\}, \mathcal{F} = \{\text{socrates}/0,\ \text{father}/1\}, \mathcal{V} = \{X, \ldots\}$

- *Terms*: X, socrates, father(socrates)
- *Atomic formulas*: mortal(X), mortal(socrates)
- *Non-Atomic formulas*: mortal(socrates) $\wedge$ mortal(father(socrates))

The syntax of first-order logic is defined as:

- Variables + function symbols $\rightarrow$ terms
- Terms + predicate symbols $\rightarrow$ formulas

Variables that are in the scope of a quantifier are **bound** variables, while those that are unquantified are called **free**. Free variables have their own values in a given formula (determined by a variable assignment of interpretation), while bound variables are placeholders that stand for the "thing " referred to within a formula.

Example

$\text{even}(x) \vee \text{odd}(x)$
$\forall x(\text{even}(x) \vee \text{odd}(x))$

Semantics

An interpretation I consists of:

- A *universe* U, that is a non-empty set that fixes the domain we are referring to in our formula
- For each n-ary function symbol, a n-ary *function* $I(f) : U^n \to U$
- For each n-ary predicate symbol, a n-ary *relation* $I(p) : U^n \to [\text{true}, \text{false}]$
- For each variable X of $\mathcal{V}$, a mapping $\eta : X \to U$, where each element of X is associated with a value in the universe U

In simpler terms, we assign meaning to function and predicate symbols by associating them with objects. These objects are function and relations into a particular domain. An atomic formula is essentially a relationship, where a predicate symbol is applied to its corresponding arguments. The relation is satisfied when the predicate's arguments belong to the interpretation of that relation.

Example

In the given interpretation, where $I(>)$ is defined as:

$$I(>) = \begin{cases} (1,0) \\ (2,1) \quad (2,0) \\ \vdots \end{cases}$$

$5 > 4$ is true because the pair (5,4) belong to $I(>)$.

Example

$\forall x\ p(x,a,b) \to q(b,x)$

$$I_1(.) = \begin{cases} U = \text{real things} & \\ a = \text{food} & b = \text{Fitz the cat} \\ p = \_ \text{ gives } \_\ \_ & q = \_ \text{ loves } \_ \end{cases}$$

I_1 : Fitz the cat loves everybody who gives him food

$$I_2(.) = \begin{cases} U = \text{natural numbers} & \\ a = 5 & b = 10 \\ p = \_ + \_ > \_ & q = \_ < \_ \end{cases}$$

I_1 : 10 is less than any X if X+5>10

Exam

Transform these sentences written in natural language into first-order logic:

1. All researchers and professors working at UNIBO have the badge.
2. There is at least a person who has not the badge.
3. There is at least a person working at UNIBO who is neither a researcher nor a professor.

We begin by identifying atomic formula that represent the basic piece of information:

- $r(X)$ means that X is a researcher (the predicate r is true whenever X is a researcher).
- $p(X)$ means that X is a professor.
- $w(X)$ means that X works at UNIBO.
- $b(X)$ means that X has the badge.

1. $\phi : \forall X((r(X) \wedge p(X)) \wedge w(X)) \to b(X)$;
2. $\psi : \exists X \neg b(X)$
3. $\xi : \exists X w(x) \wedge (\neg r(X) \wedge \neg p(X)) \equiv \exists X w(X) \wedge \neg(r(X) \vee p(X))$

Definition 19: Logical Consequence

B is a logical consequence of A if, for any interpretation I that makes A true, it also makes B true.

To prove that, one must demonstrate its validity across all possible interpretations, which are infinite.

Theorem 4: Church

While in propositional logic, an algorithm exists to verify the validity of a formula, in first-order logic determining validity is undecidable.

Instead, to prove the contrary, one must show an interpretation where the first formula is true, and the second is false.

Undecidability

- Is it possible to prove the validity of a formula through a syntactical approach similar to natural deduction?

  Even if a formula is undecidable as a logical consequence of another formula, we can still use a proof system that starts with the formulas Γ and allows us to perform some operations. In the end, this enables us to show whether the formula is true, false, or undecidable in other cases (that proof system is the basis of logic languages).

- Would it be possible to prove validity in a smaller subset of first-order logic?

  When narrowed down to a subset of first-order logic, such as first-order arithmetic, where only variables, two constants (0 and 1), the functions $+$ and $\times$, and the predicate $=$ are present, and all formulas are in form of equations, it is still undecidable (Gödel incompleteness theorem).

Exam

Starting from the previous example, is the third sentence a logical consequence of the first two?

$$\phi \wedge \psi \models \xi$$

No, we can have an interpretation where all people working at UNIBO are researchers and professors, plus some people who do not work at UNIBO.

1.4 Substitution and unification

Definition 20: Substitution

A substitution σ is a mapping $\sigma : \mathcal{V} \to \mathcal{T}$ that replaces a finite number of variables with terms (where terms, of course, can be other variables), written as:

$$\{X_1 \to t_1, \ldots, X_n \to t_n\}$$

where X_i are distinct variables and t_i are terms.

In other words, if we have an expression on e' and we apply a substitution e, the result will be a new expression based on the substituted values from e. In this case, e is an instance of $e' : e = e'\sigma$.

Example

$\sigma = \{x/f(a), y/g(z)\}$

$$h(X, b, y)\sigma = h(f(a), b, g(z))$$

Example

Substitutions follow the rules of function composition:
$\sigma = \{x/f(a), y/g(z)\}$
$\tau = \{z/c\}$

$$(h(x, y)\sigma)\tau = h(f(a), g(z))\tau = h(f(a), g(c))$$

When substitution just replaces the names of all variables, it is called variable renaming.

Example

Original expression $f(x, y)$.

- Variable renaming: $\beta = \{x/z, y/w\} \quad f(x, y)\beta = f(z, w)$
- NOT variable renaming: $\beta = \{x/z, y/z\} \quad f(x, y)\beta = f(z, z)$

The second substitution makes two different variable the same, essentially merging them.

Definition 21: Unification

A substitution σ is a unifier if, when applied to expression $e_1, \ldots, e_n$, the resulting expression is syntactical equal, i.e., $e_1\sigma = \ldots = e_n\sigma$.

Example

$f(X, a)\sigma = f(b, Y)\sigma$
Here, we can operate only on variables, not on consta I cannot modify the constants so I operate on variables
$\sigma = \{X/b, Y/a\}$

Example

$f(X, g(z))\sigma = f(b, Y)\sigma$
$\sigma_1 = \{X/b, Y/g(z)\}$
$\sigma_2 = \{X/b, Y/g(z), Z/f(a)\}$
$\vdots$

From this example, it becomes clear that there exists an infinite number of unifiers. By introducing an additional variable $\rightarrow$ term, we can have a new unifier. The *most general unifier* is the one that requires the minimum amount of substitutions, such as σ_1. Other unifiers can be derived from the most general one by composing it with additional substitutions.

Unification algorithm

- start with an empty set of substitutions ε
- scan terms simultaneously from left to right
- check if the symbols are equivalent:
  - different $\rightarrow$ failure
  - identical $\rightarrow$ continue:
    - one is unbound variable and the other term:
      - variable $\rightarrow$ failure
      - apply the substitution to the logical expression and add it to the substitution set
    - variable is not unbound: replace it by applying substitution

to unify	current substitution, remarks
$p(X, f(a)) = p(a, f(X))$	ε, start
$X := a$	$\{X \mapsto a\}$, substitution added
$f(a) := f(X)$	continue
$a := X$	$\{X \mapsto a\}$, variable is not unbound
$a := a$	continue
MGU is	$\{X \mapsto a\}$

Table 1: Unification example 1

to unify	current substitution, remarks
$p(X, f(b)) = p(a, f(X))$	ε, start
$X := a$	$\{X \mapsto a\}$, substitution added
$f(b) = f(a)$	continue
$b = a$	failure

Table 2: Unification example 2

s	t	
f	g	failure
a	a	ε
X	a	$\{X \mapsto a\}$
X	Y	$\{X \mapsto Y\}$, but also $\{Y \mapsto X\}$
$f(a, X)$	$f(Y, b)$	$\{Y \mapsto a, X \mapsto b\}$
$f(g(a, X), Y)$	$f(c, X)$	failure
$f(g(a, X), h(c))$	$f(g(a, b), \mathrm{Y})$	$\{X \mapsto b, Y \mapsto h(c)\}$
$f(g(a, X), h(Y))$	$f(g(a, b), Y)$	failure

Table 3: Unification outcomes

2 Logic Programming

A logic program is a set of axioms defining relationship between objects. A computation of a logic program is a deduction of consequences of the program.

Syntax

Atom	A, B	$::=$	$p(t_1, \ldots, t_n),\ n \geq 0$
Goal	G, H	$::=$	$\top \mid \bot \mid A \mid G \wedge H$
Clause	K	$::=$	$A \leftarrow G$
Program	P	$::=$	$K_1 \ldots K_m,\ m \geq 0$

Semantics. We define the semantics of LP in terms of a transition system.
The program begins from an initial state, and through a sequence of steps, reaches a final state. Each state represents a snapshot of computation at a specific moment. The full computation can be viewed as a sequence of steps:

$$\text{state}_1 \rightarrow \text{state}_2 \rightarrow \ldots \rightarrow \text{state}_n$$

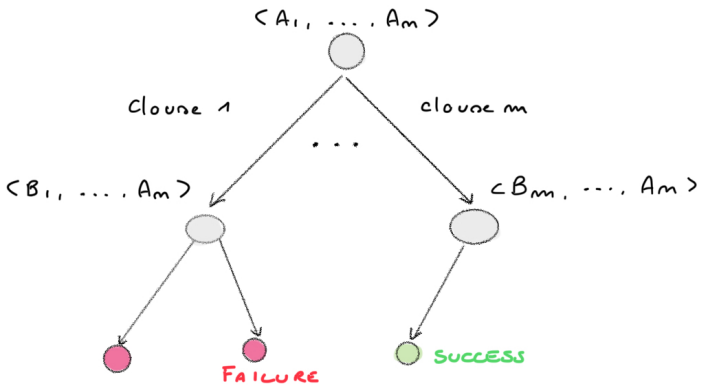
- A state consists of a pair $\langle G, \theta \rangle$: a substitution θ, expressing variable values, and a goal G, representing the remaining program to be evaluated.
- The arrow means a computational step, that transition from one state to another. This transition is defined by the syntax of the language. For every possible command, we define a corresponding transition step, moving us from one state to another. Luckily, in logic programming languages, there is only one construct, so we need only to define one transition.

2.1 Unfolding rule/Resolution

Given a goal $\langle A \wedge G, \theta \rangle$, the unfolding rule operates as follows:

1. From the goal, select an arbitrary atom A.
2. Look for a clause in the program of the form $B \leftarrow H$ such that B and A have the same predicate name.
3. Perform an unification between the selected atom A and the head of the clause B. The result is the most general unifier β that makes B and $A\theta$ identical.
4. Obtain a new state $\langle H \wedge G, \theta\beta \rangle$ by replacing A with the body H and updating the current substitutions θ with those from β, leading to a new substitution set $\theta\beta$.

Since any atom in $A \wedge G$ and any clause $B \leftarrow H$ in P for which B and $A\theta$ are unifiable can be chosen, the unfold transition is non-deterministic.
To avoid the non-determinism, a strategy is to select the leftmost atom and use all possible clauses. By doing so, we obtain a search tree where our objective is to reach the leaf that corresponds to a success case. The search is done by using DFS, starting from the topmost clause. The way we write our program is important, as DFS may get trapped in infinite loops and fail to reach the solution.



The logical view of this rule is called resolution, which is a simpler proof system than natural deduction and has only one rule:

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

where R and R' are disjunction of literals and A is an atomic formula.
The idea is to work by contradiction, i.e. to use refutation: theory united with negated consequence must be unsatisfiable.
The procedure is as follows:

1. Transform all formulas into disjunction of literals (i.e. clauses).
2. Add the negation of what we want to prove to the set of clauses.
3. Apply the resolution rule.
4. Stop when the empty clause is derived (i.e. the contradiction, i.e. false).

Example

Suppose we want to prove that B is a logical consequence of $F = \{A \to B, A\}$. We proceed as follows:

1. First, we transform formulas in F into clausal form:

$$F = \{\neg A \vee B, A\}$$

2. Then, we add negation of what we want to prove to F

$$F = \{\neg A \vee B, A, \neg B\}$$

3. We apply resolution:

$$\dfrac{\dfrac{\neg A \vee B \quad A}{B} \qquad \neg B}{\square}$$

4. Having obtained the empty clause, we conclude that F is inconsistent and, therefore $F \models B$.

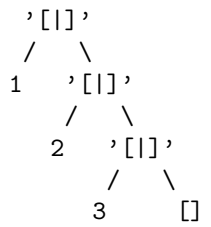
What we have shown applies to resolution for propositional logic; for FOL, the idea remains the same. However, transforming FOL formulas into clauses is more complex, and the resolution rule involves unification due to the presence of variables. While resolution is sound and complete for both propositional logic and FOL, the order in which clauses are selected during resolution can, in some instances, either prevent us from obtaining the empty clause or result in indefinite creation of new clauses, never reaching the empty clause. This shows the semi-decidability in FOL.

2.2 Prolog

In Prolog there is no iteration, but you can get iterative behaviour through recursion.

Syntax

- Variables: `X,Y` (Upper case initial)
- Constants: `alice,a` (Lower case initial)
- Functions: `father(X)`
- Predicates: `brother(X,Y)`
- Facts: `head.`
- Rules: `head :- body.`
- Lists: A recursive data structure, based on the concept of cons lists, that is either empty or consists of two parts: a head and a tail (has to be a list). The `|` separates the head and the tail parts.



The following are lists:

```
[] empty list
[a | [b | []]] or [a, b]
[a | [b | [c | []]]] or [a, b, c]
[[] | []] or [[]]
[[a | []] | [b | []]] or [[a], b]
```

– **member**

```
% base case
member(X, [X | _]) % first element
% recursively check on the tail
member(X, [_ | Tail]) :- member(X, tail)
```

– **length**

```
% base case
length([], 0).
% _ anonymous variable
length([_ | Tail], N) :- length(Tail, NT), N is NT+1.
```

– **append**

```
% base case
append([], L, L)
append([H | T], L2, [H | NewTail]) :-
    append(Rest1, L2, NewTail)
```

* the recursion case, which reduces the first list by one element and attempts to append `L2` to it again. This process repeats until the first list is empty, triggering the base case.
* as the recursion unwinds, `NewTail` is built up backward.

Example

```
?- append([1, 2, 3], [4, 5, 6], Result).
Result = [1, 2, 3, 4, 5, 6].

% Breaking down the recursion:
append([1, 2, 3], [4, 5, 6], Result).
H = 1, T = [2, 3], L2 = [4, 5, 6]
append([2, 3], [4, 5, 6], NewTail).
H = 2, T = [3], L2 = [4, 5, 6]
append([3], [4, 5, 6], NewTail).
H = 3, T = [], L2 = [4, 5, 6]
% Base case reached, NewTail is [4, 5, 6]
append([], [4, 5, 6], NewTail).
```

In imperative programming languages, the position of a variable in an assignment operation matters: if a variable is on the right side, its value is accessed, and if it's on the left side, a value is assigned to it. This means you cannot use the same operation for both input and output simultaneously. However, in Prolog, it's possible to compute arguments as both input and output depending on how the call is specified. This flexibility is due to Prolog use of unification, which is **bidirectional**, unlike the one-way assignment in imperative languages.

Arithmetic. In Prolog, integers and floating-point numbers are constants. There are several math operators predefined as function symbols (`+,-,*, div,mod`).
To evaluate arithmetic expressions, Prolog uses the predicate `X is Expression`, where `X` will be instantiated to the value of `Expression`.

Example

`X is 2 + 3` will evaluate the expression 2 + 3 and unify X with the result, 5.

Variables in `Expr` must be already instantiated at the moment of evaluation.

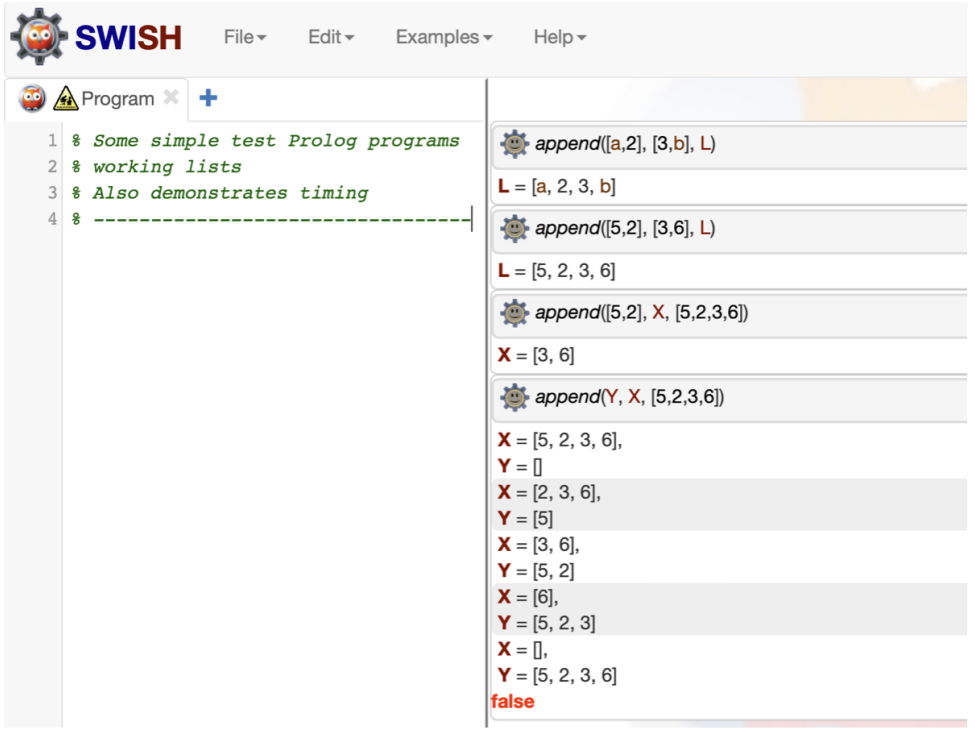


Figure 1: Bidirectional argument passing

Example

```
p(X) :- Y is 3+5, X is Y.
?- p(X).
X=8

p(X):-X is Y,Y is 3+5.
?- p(X).
error
```

Expressions can be compared using relational operators such as `=:=,=;>,<,>=,=<` to evaluate their values.

Example

Count the number of leaves of binary tree.

```
  P <-     P <-     P <- Internal nodes  X <- Leaf
 / \      / \      / \                   / \
L   R    L  nil   nil R                nil nil
```

First of all, how we represent a binary tree? I can use a function symbol, `tree(P, L, R)`, and so can be used as term for another predicate. Now, I want to construct a predicate which given a tree counts the number of leaves of tree.

```
tree_count(nil, 0)
tree_count(tree(_, nil, nil), 1)
tree_count(tree(_, L, R), N) :-
    tree_count(L, N1),
    tree_count(R, N2),
    N is N1 + N2
```

The predicate ! (cut). The search strategy of the Prolog interpret attempts to inspect the entire search tree. To modify this behavior, we can use the predicate cut (`!`) which can be inserted into the body of any clause. When evaluated, it make some choices as definite and non-backtrackable, thus allowing us to prune part of the search tree.

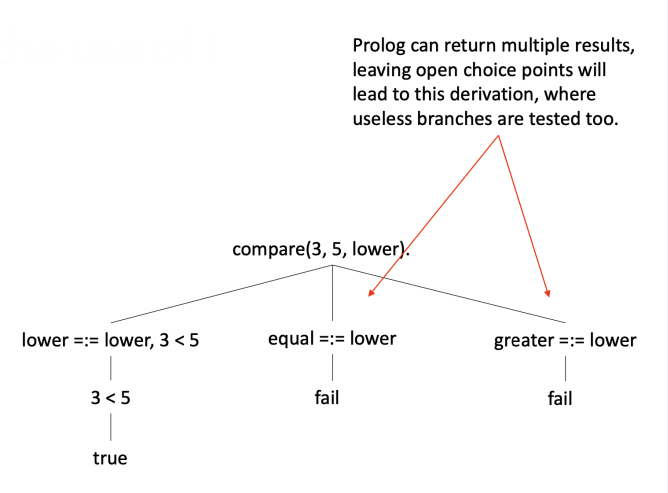
Example

Consider the clause:

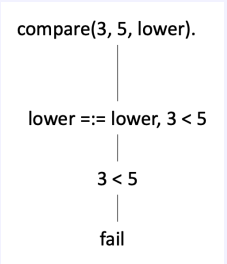
```
p :- q, r !, s.
```

The choices made in the evaluation of goal `q` are definite. If `s` fails, Prolog will not backtrack to try to find another solution for either `q` or `r` but will fail immediately.

Example

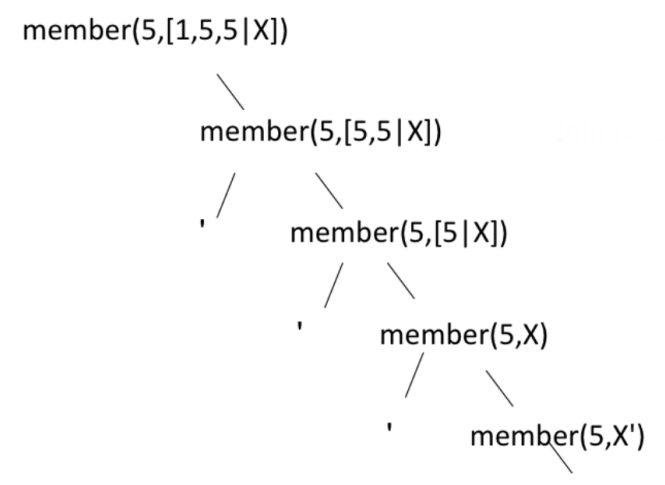



```
compare(X, Y, lower) :- X < Y.
compare(X, Y, equal) :- X =:= Y.
compare(X, Y, greater) :- X > Y.

?- compare(3, 5, lower).
```



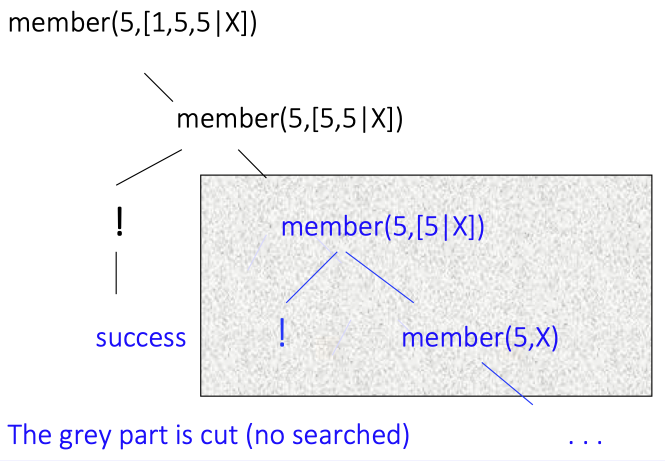

```
compare(X, Y, lower) :- X < Y, !.
compare(X, Y, equal) :- X =:= Y, !.
compare(X, Y, greater) :- X > Y.

?- compare(3, 5, lower).
```

Example




```
member(X, [X|_]).
member(X,[-|L]) :- member(X,L).
```

Each time Prolog finds a 5, and since `X` is an unbound variable, it means Prolog explores an infinite number of possible lists where 5 is a member, leading to an infinite number of solutions.




```
member(X, [X|_]) :- !.
member(X,[-|L]) :- member(X,L).
```

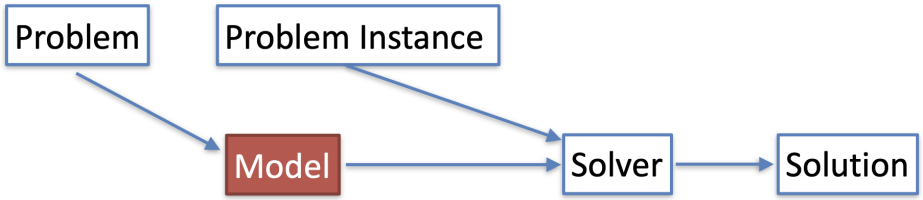
Example

The following code shows how to achieve the logic of `if then else` using the cut operator:

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

3 Constraint programming

In constraint programming, the goal is to create a model of the problem through the use of a predefined set of constraints (equations). These constraints can be either global, supplied by external sources, or custom, defined by the programmer themselves. Once the problem has been modeled, a predefined solver is employed to find a solution for us.



There are two main classes of problems:

Definition 22: Constraint Satisfaction Problems (CSP)

A CSP is defined by:

- a finite set of variables $\{X_1, \ldots, X_n\}$
- a set of possible values (Domain) for these variables $D(X_1), \ldots, D(X_n)$
- a set of constraints $\{C_1, \ldots, C_n\}$

A solution to a CSP is an assignment to all the variables that satisfies the constraints.

Similar to how a predicate is evaluated under an interpretation in FOL, a constraint is satisfied if a pair of variables meets a defined meaning. Specifically, the meaning of a constraint is given by specifying a set of values that satisfy the relation. If a pair of variables belongs within this set, the constraint is thereby satisfied.

Definition 23: Constraint Optimization Problems (COP)

A COP is a CSP plus a function f which expresses some cost. A solution to a COP is those values that optimizes this cost function f.

We have two main families of constraint languages, to which we add constraints (and the relative solvers):

- **Constraint Logic Programming**
- **Imperative languages with constraints**

3.1 Constraint Logic Programming

Definition 24: Alphabet

The alphabet is an extension of that used in logic programming 1.3, augmented with symbols specific to constraints. Within this framework, a constraint can be seen as:

- an atomic constraint, which is an expression $c(t_1, \ldots, t_n)$ with $n \geq 0$ terms as its arguments.
- a conjuction of constraints.

Additionally, CLP supports *First-Order Constraint Theory*, which provides the possibility of defining the semantics of a constraint by allowing the programmer to introduce a set of axioms.

Syntax

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	$::=$	$K_1 \ldots K_m,\ m \geq 0$

In simpler terms, in the body of the clauses we can have constraints.

Example

"Send+More=Money" is puzzle where each letter represents a unique digit from 0 to 9. The goal is to find a digit for each letter such that when you add the numbers SEND and MORE together, they equal the number MONEY.

```
[S,E,N,D,M,O,R,Y] = [9,5,6,7,1,0,8,2]

          SEND
          9567
      +   MORE
          1085
         -----
      = MONEY
        10652
```

```
:- use_module(library(clpfd)).

send([S,E,N,D,M,O,N,E,Y]) :-
    gen_domains([S,E,N,D,M,O,N,E,Y], 0..9),
    S #\=0, M #\= 0,
    all_distinct([S,E,N,D,M,O,R,Y]),
        1000*S + 100*E + 10*N + D
        + 1000*M + 100*O + 10*R + E
        #= 10000*M + 1000*O + 100*N + 10*E + Y,
    labeling([],[S,E,N,D,M,O,R,Y]).

gen_domains([],_).
gen_domains([H|T],D) :- H in D, gen_domains(T,D).
```

- Check each element in the list to take a value from 0 to 9;
- Check that M and S are different from 0;
- Check all variables are distinct from one another;
- Check the sum holds, considering the positional encoding for each digit.
- Ground variables with values;

The above example might seem Prolog, but not exactly. In pure Prolog, as we have said, variables must be grounded with arithmetic Prolog built-ins, so we cannot proceed as described in the code. We must move the `labeling` line below `gen_domains`. However, doing so means that every time one of these conditions does not hold, it backtracks to find another solution through different variable choices. This approach is impracticable for our problem.
Luckily, the `clpfd` library allows us to use constraints in Prolog and provides a substitute for Prolog arithmetic built-ins by simply prefixing them with a `#`. In this case, we are allowed to use constraints on variables that are not grounded and structure the program as described above. The key point here is to first accumulate the constraints and afterward ground the variables with values. The constraints prune the search tree before actively do the search. In other words, they limit the number of branches that could exist. Thus, a problem that may seem impracticable can become easy to resolve.

Semantics. Constraint Logic Programming has a transition system similar to the one observed in logic programming, as previously discussed (2.1), with some minor differences:

- **Unfold**: Given $\langle A \wedge G, C\rangle$, the result of computation here is represented by a constraint rather than a substitution. We select a compatible clause with a matching predicate name and unifiable $B \leftarrow H$, from program P. In the unfold step $\langle A \wedge G, C\rangle \mapsto \langle H \wedge G, (B = A) \wedge C\rangle$, we replace the call with the body of the clause as usual. However, instead of performing unification directly, we add equations representing the constraint that perform the unification implicitly. By solving these equations, we achieve the same result as the unification, even though these equations are not solved.
- **Solve**: we simplify the constraints accumulated so far by combining them and applying them to narrow down the possible values of the variables $(C \wedge D_1) \leftrightarrow D_2$.

3.2 MiniZinc

Syntax

- Each expression terminates with `;`
- Variable domain and array index domain must all be specified
- Index array starts from 1 not 0
- Comments `%`
- Range `0..10`
- Arithmetic Operators `+,-,*,/,^,=,!=`
- Logical Operators `\/,/\,->,!`

- We specify data for parameters in separate files ending with `.dzn` extension. To execute the program, we can use the following command:

  ```
  minizinc program.mzn data.dzn
  ```

MiniZinc allows the programmer to specify:

- **Parameters**: we instantiate the values of these based on the current problem instance.

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

- **Variables**: variables that will be instantiated by the solver.

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

- **Constraints**: rules that a solution must respect.

  ```
  constraint [expression]
  ```

  Global constraints:

  ```
  include "global.mzn";
  all_different()
  all_equal()
  ```

- **Arrays**: can be either a set of parameters or variables.

  ```
  array[index_domain] of [domain]
  array[index_domain] of var [domain]
  ```

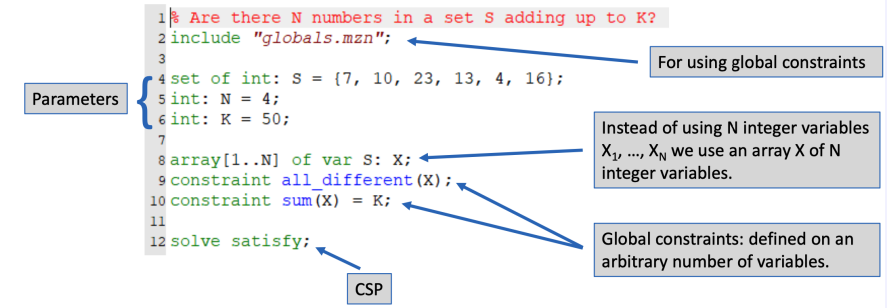
- **Aggregation functions**: `sum, product, min, max, forall, exists`

  ```
  % sum of all the elements in array
  sum(array_x)
  % sum of elements from 1 to 3 of array
  sum(i in 1..3)(array_x[i])
  % first 3 elements in array are different
  forall (i,j in 1..3 where i < j) (array_x[i] != array_x[j])
  ```

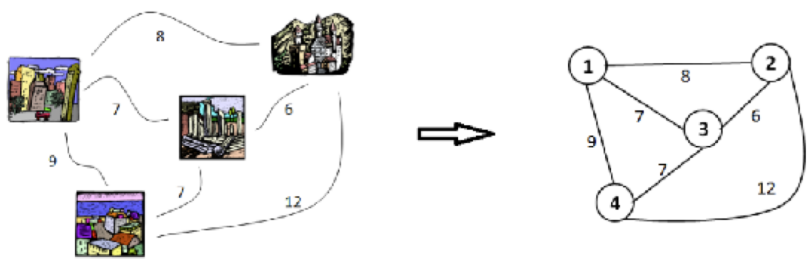
A MiniZinc program includes `solve` keyword, followed by `satisfy`, `maximize`, or `minimize` to indicate the kind of problem. For the cases of `maximize` and `minimize`, an objective function must be also specified. Without `solve` keyword, it is implicitly a satisfaction problem.

Example

Subset-sum problem: are there N numbers in a set S adding up to K?



Example



Traveling Salesman Problem:
I visit each city once and I want to save my time

```
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);

var int: total_distance = sum(i in 2..n)(dist[city[i-1],city[i]]);
solve minimize total_distance;

output [city_name[fix(city[i])] ++ " -> " | i in 1..n ] ++
        ["\nTotal hours travelled: ", show(total_distance) ];

```