An introduction to Prolog

Pedro Cabalar

Department of Computer Science
University of Corunna, SPAIN

January 31, 2023

A first glimpse at Prolog

- PROLOG stands for "PROgramming in LOGic" (originally in French "*PROgrammation en LOGique*").

A first glimpse at Prolog

- PROLOG stands for "PROgramming in LOGic" (originally in French "*PROgrammation en LOGique*").
- Well suited for symbolic, non-numeric computation. Good for dealing with objects and relations.

A first glimpse at Prolog

- PROLOG stands for "PROgramming in LOGic" (originally in French "*PROgrammation en LOGique*").
- Well suited for symbolic, non-numeric computation. Good for dealing with objects and relations.
- Let us start with facts (ground atoms) for some relations (predicates).

Typical example: family relationships

- Example: Juan Carlos is the father of Felipe, Cristina and Elena.

Typical example: family relationships

- Example: Juan Carlos is the father of Felipe, Cristina and Elena. This can be expressed by the following three facts:

```
father(juancarlos,felipe).
father(juancarlos,cristina).
father(juancarlos,elena).
```

Typical example: family relationships

- Example: Juan Carlos is the father of Felipe, Cristina and Elena. This can be expressed by the following three facts:

```
father(juancarlos,felipe).
father(juancarlos,cristina).
father(juancarlos,elena).
```

Their mother is Sofia:

```
mother(sofia,felipe).
mother(sofia,cristina).
mother(sofia,elena).
```

Typical example: family relationships

- Example: Juan Carlos is the father of Felipe, Cristina and Elena. This can be expressed by the following three facts:

```
father(juancarlos,felipe).
father(juancarlos,cristina).
father(juancarlos,elena).
```

  Their mother is Sofia:

```
mother(sofia,felipe).
mother(sofia,cristina).
mother(sofia,elena).
```

  Felipe and Letizia have two children:

```
father(felipe,leonor).
father(felipe,sofia2).
mother(letizia,leonor).
mother(letizia,sofia2).
```

Typical example: family relationships

- We can query these facts as a relational data base. Is Juan Carlos, Elena's father? Is he Sofia's father?

```
?- father(juancarlos,elena).
?- father(juancarlos,sofia).
```

Typical example: family relationships

- We can query these facts as a relational data base. Is Juan Carlos, Elena's father? Is he Sofia's father?

```
?- father(juancarlos,elena).
?- father(juancarlos,sofia).
```

- Queries may contain variables (identifiers beginning with capital letters).

Typical example: family relationships

- We can query these facts as a relational data base. Is Juan Carlos, Elena's father? Is he Sofia's father?

```
?- father(juancarlos,elena).
?- father(juancarlos,sofia).
```

- Queries may contain variables (identifiers beginning with capital letters). Solutions = instantiations of variables for which the answer is `Yes`.

Typical example: family relationships

- We can query these facts as a relational data base. Is Juan Carlos, Elena's father? Is he Sofia's father?

```
?- father(juancarlos,elena).
?- father(juancarlos,sofia).
```

- Queries may contain variables (identifiers beginning with capital letters). Solutions = instantiations of variables for which the answer is `Yes`.
- We type ';' to find more answers or return to stop.

Typical example: family relationships

- We can query these facts as a relational data base. Is Juan Carlos, Elena's father? Is he Sofia's father?

```
?- father(juancarlos,elena).
?- father(juancarlos,sofia).
```

- Queries may contain variables (identifiers beginning with capital letters). Solutions = instantiations of variables for which the answer is `Yes`.
- We type ';' to find more answers or return to stop.
- What would these queries mean?

```
?- father(X,leonor).
?- mother(sofia,X).
?- father(X,Y).
?- mother(X,Y), father(Y,leonor).
```

Typical example: family relationships

- We can query these facts as a relational data base. Is Juan Carlos, Elena's father? Is he Sofia's father?

```
?- father(juancarlos,elena).
?- father(juancarlos,sofia).
```

- Queries may contain variables (identifiers beginning with capital letters). Solutions = instantiations of variables for which the answer is `Yes`.
- We type ';' to find more answers or return to stop.
- What would these queries mean?

```
?- father(X,leonor).
?- mother(sofia,X).
?- father(X,Y).
?- mother(X,Y), father(Y,leonor).
```

- How would you check whether Cristina and Elena have the same father?

Typical example: family relationships

- The comma means conjunction. These queries are logically equivalent:

```
?- mother(X,Y), father(Y,leonor).
?- father(Y,leonor), mother(X,Y).
```

  although their computation is different, as we will see later.

Typical example: family relationships

- The comma means conjunction. These queries are logically equivalent:

```
?- mother(X,Y), father(Y,leonor).
?- father(Y,leonor), mother(X,Y).
```

  although their computation is different, as we will see later.

- Sometimes, a variable is irrelevant. We can use '_' to ignore its value in the answer. Example: is Felipe a father?

```
?- father(felipe,_).
```

Typical example: family relationships

- The comma means conjunction. These queries are logically equivalent:

  ```
  ?- mother(X,Y), father(Y,leonor).
  ?- father(Y,leonor), mother(X,Y).
  ```

  although their computation is different, as we will see later.

- Sometimes, a variable is irrelevant. We can use '_' to ignore its value in the answer. Example: is Felipe a father?

  ```
  ?- father(felipe,_).
  ```

  Who has a father and a mother?

Typical example: family relationships

- The comma means conjunction. These queries are logically equivalent:

```
?- mother(X,Y), father(Y,leonor).
?- father(Y,leonor), mother(X,Y).
```

  although their computation is different, as we will see later.

- Sometimes, a variable is irrelevant. We can use '_' to ignore its value in the answer. Example: is Felipe a father?

```
?- father(felipe,_).
```

  Who has a father and a mother?

```
?- father(_,X),mother(_,X).
```

Typical example: family relationships

- The comma means conjunction. These queries are logically equivalent:

  ```
  ?- mother(X,Y), father(Y,leonor).
  ?- father(Y,leonor), mother(X,Y).
  ```

  although their computation is different, as we will see later.

- Sometimes, a variable is irrelevant. We can use '_' to ignore its value in the answer. Example: is Felipe a father?

  ```
  ?- father(felipe,_).
  ```

  Who has a father and a mother?

  ```
  ?- father(_,X),mother(_,X).
  ```

  Notice that the two '_' are different irrelevant variables.

Adding rules

- We can "give name" to queries using rules. For instance, for:

```
?- mother(X,Y), father(Y,Z).
```

we can define a new predicate `grandmother` using:

```
grandmother(X,Z) :- mother(X,Y), father(Y,Z).
```

Adding rules

- We can "give name" to queries using rules. For instance, for:

```
?- mother(X,Y), father(Y,Z).
```

we can define a new predicate `grandmother` using:

$$\underbrace{\texttt{grandmother(X,Z)}}_{\text{Rule head}}\ \texttt{:-}\ \underbrace{\texttt{mother(X,Y), father(Y,Z).}}_{\text{Rule body}}$$

Adding rules

- We can "give name" to queries using rules. For instance, for:

```
?- mother(X,Y), father(Y,Z).
```

  we can define a new predicate `grandmother` using:

$$\underbrace{\texttt{grandmother(X,Z)}}_{\text{Rule head}}\ \texttt{:-}\ \underbrace{\texttt{mother(X,Y), father(Y,Z).}}_{\text{Rule body}}$$

- The ':–' symbol is read as if or as a $\leftarrow$ implication.

Adding rules

- We can "give name" to queries using rules. For instance, for:

```
?- mother(X,Y), father(Y,Z).
```

we can define a new predicate `grandmother` using:

$$\underbrace{\texttt{grandmother(X,Z)}}_{\text{Rule head}}\ \texttt{:-}\ \underbrace{\texttt{mother(X,Y), father(Y,Z).}}_{\text{Rule body}}$$

- The ':−' symbol is read as if or as a $\leftarrow$ implication.
- In first order logic, we would write:

$$\forall x \forall y \forall z \; \big(\, Mother(x,y) \wedge Father(y,z) \rightarrow Grandmother(x,z) \, \big)$$

Adding rules

- We can "give name" to queries using rules. For instance, for:

```
?- mother(X,Y), father(Y,Z).
```

  we can define a new predicate `grandmother` using:

$$\underbrace{\texttt{grandmother(X,Z)}}_{\text{Rule head}}\ \texttt{:-}\ \underbrace{\texttt{mother(X,Y), father(Y,Z).}}_{\text{Rule body}}$$

- The '`:-`' symbol is read as if or as a $\leftarrow$ implication.
- In first order logic, we would write:

$$\forall x \forall y \forall z \ \left(\ Mother(x,y) \wedge Father(y,z) \rightarrow Grandmother(x,z)\ \right)$$

- We can use it now in queries:

```
?- grandmother(X,leonor).
```

Adding rules

- We may have several rules to define a predicate.

Adding rules

- We may have several rules to define a predicate. For instance, my mother's mother is also my grandmother:

```
grandmother(X,Z) :- mother(X,Y), father(Y,Z).
grandmother(X,Z) :- mother(X,Y), mother(Y,Z).
```

  to obtain solutions to `?- grandmother(X,Y).` we can apply any of these rules.

Adding rules

- Another example: define the `parent` relation.

Adding rules

- Another example: define the `parent` relation.

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Adding rules

- Another example: define the `parent` relation.

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

  We can use disjunction '`;`' for rules with same head

```
parent(X,Y) :- father(X,Y) ; mother(X,Y).
```

Adding rules

- Another example: define the `parent` relation.

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

  We can use disjunction '`;`' for rules with same head

```
parent(X,Y) :- father(X,Y) ; mother(X,Y).
```

- Exercises: who are Felipe's parents? Redefine `grandmother` with a single rule using the `parent` relation.

Adding rules

- Some predicates never occur in rule heads, excepting facts. These are called extensional.

Adding rules

- Some predicates never occur in rule heads, excepting facts. These are called extensional.
- But of course, a predicate may combine rules and facts.

Adding rules

- Some predicates never occur in rule heads, excepting facts. These are called extensional.
- But of course, a predicate may combine rules and facts. For instance, predicate `female`, we may include some facts

```
female(cristina).  female(elena).
female(leonor).  female(sofia2).
```

Adding rules

- Some predicates never occur in rule heads, excepting facts. These are called extensional.
- But of course, a predicate may combine rules and facts. For instance, predicate `female`, we may include some facts

  ```
  female(cristina).  female(elena).
  female(leonor).  female(sofia2).
  ```

  but we can also derive it from `mother`

  ```
  female(X) :- mother(X,_).
  ```

Adding rules

- Exercise: define the `sister` relation.

Adding rules

- Exercise: define the `sister` relation.

```
sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X).
```

```
?- sister(felipe,X).
?- sister(leonor,X).
```

Adding rules

- Exercise: define the `sister` relation.

```
sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X).
```

```
?- sister(felipe,X).
?- sister(leonor,X).
```

- Problem: Leonor is sister of herself! We should specify that they are different:

```
sister(X,Y) :- parent(Z,X),parent(Z,Y),
               female(Y), X \= Y.
```

Recursion

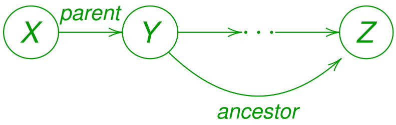
- Rules can be recursive, that is, a head predicate may also occur in the body.

Recursion

- Rules can be recursive, that is, a head predicate may also occur in the body.
- For instance, define the `ancestor` relation as the transitive closure of `parent`:

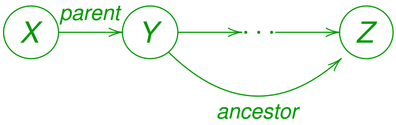
Recursion

- Rules can be recursive, that is, a head predicate may also occur in the body.
- For instance, define the `ancestor` relation as the transitive closure of `parent`:



Recursion

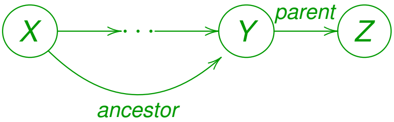
- Rules can be recursive, that is, a head predicate may also occur in the body.
- For instance, define the `ancestor` relation as the transitive closure of `parent`:




```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

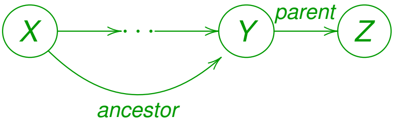
Recursion

- Another alternative can be:



Recursion

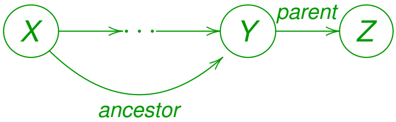
- Another alternative can be:




```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(Y,Z), ancestor(X,Y).
```

Recursion

- Another alternative can be:



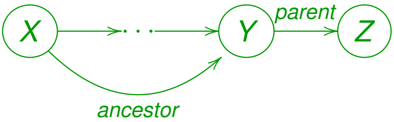

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(Y,Z), ancestor(X,Y).
```

- In principle, this program is equivalent to:

```
ancestor(X,Z) :- ancestor(X,Y), parent(Y,Z).
ancestor(X,Y) :- parent(X,Y).
```

Recursion

- Another alternative can be:




```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(Y,Z), ancestor(X,Y).
```

- In principle, this program is equivalent to:

```
ancestor(X,Z) :- ancestor(X,Y), parent(Y,Z).
ancestor(X,Y) :- parent(X,Y).
```

but Prolog further introduces an evaluation ordering that, for instance, causes query `?- ancestor(X,juancarlos)` to iterate forever.

Top-down goal satisfaction

- So, how does this work? Take

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Top-down goal satisfaction

- So, how does this work? Take

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

- The query `?- ancestor(sofia,leonor).` fixes a first goal.

Top-down goal satisfaction

- So, how does this work? Take

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

- The query `?- ancestor(sofia,leonor).` fixes a first goal. Prolog will look for rule heads that match the current goal.

Top-down goal satisfaction

- So, how does this work? Take

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

- The query `?- ancestor(sofia,leonor).` fixes a first goal. Prolog will look for rule heads that match the current goal.
- For instance, the first rule matches under the replacement `X=sofia`, `Y=leonor`.

Top-down goal satisfaction

- So, how does this work? Take

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

- The query `?- ancestor(sofia,leonor).` fixes a first goal. Prolog will look for rule heads that match the current goal.
- For instance, the first rule matches under the replacement `X=sofia`, `Y=leonor`. This is like having the rule instance:

```
ancestor(sofia,leonor) :- parent(sofia,leonor).
```

Top-down goal satisfaction

- So, how does this work? Take

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

- The query `?- ancestor(sofia,leonor).` fixes a first goal. Prolog will look for rule heads that match the current goal.
- For instance, the first rule matches under the replacement `X=sofia`, `Y=leonor`. This is like having the rule instance:

```
ancestor(sofia,leonor) :- parent(sofia,leonor).
```

- As matching succeeded, we replace our initial goal by the rule body `parent(sofia,leonor)`, which becomes our new goal.

Top-down goal satisfaction

- We try then to match `parent(sofia,leonor)` with some rule head. This predicate has two rules

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Top-down goal satisfaction

- We try then to match `parent(sofia,leonor)` with some rule head. This predicate has two rules

```
parent(sofia,leonor) :- father(sofia,leonor).
parent(X,Y) :- mother(X,Y).
```

- The first one matches, so our new goal becomes `father(sofia,leonor).`

Top-down goal satisfaction

- We try then to match `parent(sofia,leonor)` with some rule head. This predicate has two rules

```
parent(sofia,leonor) :- father(sofia,leonor).
parent(X,Y) :- mother(X,Y).
```

- The first one matches, so our new goal becomes `father(sofia,leonor)`.
- However, `father` is extensional (only facts), and this fact is not included in the program. So, our goal fails.

Top-down goal satisfaction

- We try then to match `parent(sofia,leonor)` with some rule head. This predicate has two rules

```
parent(sofia,leonor) :- father(sofia,leonor).
parent(X,Y) :- mother(X,Y).
```

- The first one matches, so our new goal becomes `father(sofia,leonor).`
- However, `father` is extensional (only facts), and this fact is not included in the program. So, our goal fails.
- A failure implies backtracking to the last matching, and looking for new matches.

Top-down goal satisfaction

- So we "reconsider" the last deleted goal `parent(sofia,leonor)` and try to match another rule

```
parent(X,Y) :- father(X,Y).  (failed)
parent(X,Y) :- mother(X,Y).
```

Top-down goal satisfaction

- So we "reconsider" the last deleted goal `parent(sofia,leonor)` and try to match another rule

```
parent(X,Y) :- father(X,Y). (failed)
parent(sofia,leonor) :- mother(sofia,leonor).
```

- Our new goal becomes `mother(sofia,leonor)`.

Top-down goal satisfaction

- So we "reconsider" the last deleted goal `parent(sofia,leonor)` and try to match another rule

```
parent(X,Y) :- father(X,Y).  (failed)
parent(sofia,leonor) :- mother(sofia,leonor).
```

- Our new goal becomes `mother(sofia,leonor)`. But this also fails: `mother` is extensional and this is not a fact.

Top-down goal satisfaction

- So we "reconsider" the last deleted goal `parent(sofia,leonor)` and try to match another rule

```
parent(X,Y) :- father(X,Y).  (failed)
parent(sofia,leonor) :- mother(sofia,leonor).
```

- Our new goal becomes `mother(sofia,leonor)`. But this also fails: `mother` is extensional and this is not a fact.
- Now, `parent(sofia,leonor)` has failed in its turn. We backtrack to `ancestor(sofia,leonor)` looking for another matching head.

Top-down goal satisfaction

- So we "reconsider" the last deleted goal `parent(sofia,leonor)` and try to match another rule

```
parent(X,Y) :- father(X,Y).  (failed)
parent(sofia,leonor) :- mother(sofia,leonor).
```

- Our new goal becomes `mother(sofia,leonor)`. But this also fails: `mother` is extensional and this is not a fact.
- Now, `parent(sofia,leonor)` has failed in its turn. We backtrack to `ancestor(sofia,leonor)` looking for another matching head.

```
ancestor(X,Z) :-
        parent(X,Y), ancestor(Y,Z).
```

Top-down goal satisfaction

- So we “reconsider” the last deleted goal `parent(sofia,leonor)` and try to match another rule

```
parent(X,Y) :- father(X,Y). (failed)
parent(sofia,leonor) :- mother(sofia,leonor).
```

- Our new goal becomes `mother(sofia,leonor)`. But this also fails: `mother` is extensional and this is not a fact.
- Now, `parent(sofia,leonor)` has failed in its turn. We backtrack to `ancestor(sofia,leonor)` looking for another matching head.

```
ancestor(sofia,leonor) :-
        parent(sofia,Y), ancestor(Y,leonor).
```

Top-down goal satisfaction

- Now we have a list of goals `parent(sofia,Y), ancestor(Y,leonor).`

Top-down goal satisfaction

- Now we have a list of goals `parent(sofia,Y), ancestor(Y,leonor).`
- Matching `parent(sofia,Y)` with `parent(X',Y') :- father(X',Y').` is possible under replacement `X'=sofia`, `Y'=Y`.

Top-down goal satisfaction

- Now we have a list of goals `parent(sofia,Y), ancestor(Y,leonor).`
- Matching `parent(sofia,Y)` with `parent(X',Y') :- father(X',Y').` is possible under replacement `X'=sofia, Y'=Y`. This leads to new goal `father(sofia,Y)` that fails.

Top-down goal satisfaction

- Now we have a list of goals `parent(sofia,Y), ancestor(Y,leonor).`
- Matching `parent(sofia,Y)` with `parent(X',Y') :- father(X',Y').` is possible under replacement `X'=sofia, Y'=Y`. This leads to new goal `father(sofia,Y)` that fails.
- Matching `parent(sofia,Y)` with `parent(X',Y') :- mother(X',Y').` leads to new goal `mother(sofia,Y)` that succeeds for `Y=felipe` (more matchings are possible).

Top-down goal satisfaction

- Now we have a list of goals `parent(sofia,Y), ancestor(Y,leonor).`
- Matching `parent(sofia,Y)` with `parent(X',Y') :- father(X',Y').` is possible under replacement `X'=sofia, Y'=Y`. This leads to new goal `father(sofia,Y)` that fails.
- Matching `parent(sofia,Y)` with `parent(X',Y') :- mother(X',Y').` leads to new goal `mother(sofia,Y)` that succeeds for `Y=felipe` (more matchings are possible).
- Important: assignment `Y=felipe` affects our whole list of goals.

Top-down goal satisfaction

- Now we have a list of goals `parent(sofia,Y), ancestor(Y,leonor).`
- Matching `parent(sofia,Y)` with `parent(X',Y') :- father(X',Y').` is possible under replacement `X'=sofia`, `Y'=Y`. This leads to new goal `father(sofia,Y)` that fails.
- Matching `parent(sofia,Y)` with `parent(X',Y') :- mother(X',Y').` leads to new goal `mother(sofia,Y)` that succeeds for `Y=felipe` (more matchings are possible).
- Important: assignment `Y=felipe` affects our whole list of goals. That is, `ancestor(Y,leonor)` becomes `ancestor(felipe,leonor).`

Top-down goal satisfaction

- Matching `ancestor(felipe,leonor)` with `ancestor(X,Y) :- parent(X,Y).` leads to goal `parent(felipe,leonor)`.

Top-down goal satisfaction

- Matching `ancestor(felipe,leonor)` with `ancestor(X,Y) :- parent(X,Y).` leads to goal `parent(felipe,leonor)`.
- Finally, matching `parent(felipe,leonor)` with `parent(X,Y) :- father(X,Y).` leads to new goal `father(felipe,leonor)` that succeeds.

Top-down goal satisfaction

- Matching `ancestor(felipe,leonor)` with `ancestor(X,Y) :- parent(X,Y).` leads to goal `parent(felipe,leonor)`.
- Finally, matching `parent(felipe,leonor)` with `parent(X,Y) :- father(X,Y).` leads to new goal `father(felipe,leonor)` that succeeds. Prolog answers `Yes`!

Adding functions

- We can use function symbols to pack some data together as a single structure. Example:

```
born(juancarlos,f(5,1,1938)).
born(felipe,f(30,1,1968)).
born(letizia,f(15,9,1972)).
born(sofia,f(2,11,1938)).

later(f(_,_,Y), f(_,_,Y1)) :- Y>Y1.
later(f(_,M,Y), f(_,M1,Y)) :- M>M1.
later(f(D,M,Y), f(D1,M,Y)) :- D>D1.

birthday(X,d(D,M)) :- born(X,f(D,M,_)).
```

Adding functions

- We can use function symbols to pack some data together as a single structure. Example:

```
born(juancarlos,f(5,1,1938)).
born(felipe,f(30,1,1968)).
born(letizia,f(15,9,1972)).
born(sofia,f(2,11,1938)).

later(f(_,_,Y), f(_,_,Y1)) :- Y>Y1.
later(f(_,M,Y), f(_,M1,Y)) :- M>M1.
later(f(D,M,Y), f(D1,M,Y)) :- D>D1.

birthday(X,d(D,M)) :- born(X,f(D,M,_)).
```

- Predicate $>$ is predefined for arithmetic values.

Adding functions

Some examples of queries:

- Which is Juan Carlos' date of birth?

  ```
  ?- born(juancarlos,X).
  ```

Adding functions

Some examples of queries:

- Which is Juan Carlos' date of birth?

  ```
  ?- born(juancarlos,X).
  ```

- Is Felipe older than Letizia?

  ```
  ?- born(felipe,X),born(letizia,Y),later(Y,X).
  ```

Adding functions

Some examples of queries:

- Which is Juan Carlos' date of birth?

  ```
  ?- born(juancarlos,X).
  ```

- Is Felipe older than Letizia?

  ```
  ?- born(felipe,X),born(letizia,Y),later(Y,X).
  ```

- Find two people that were born in the same year

Adding functions

Some examples of queries:

- Which is Juan Carlos' date of birth?

  ```
  ?- born(juancarlos,X).
  ```

- Is Felipe older than Letizia?

  ```
  ?- born(felipe,X),born(letizia,Y),later(Y,X).
  ```

- Find two people that were born in the same year

  ```
  ?- born(X,f(_,_,Y)),born(Z,f(_,_,Y)),X\=Z.
  ```

Adding functions

Some examples of queries:

- Which is Juan Carlos' date of birth?
  `?- born(juancarlos,X).`
- Is Felipe older than Letizia?
  `?- born(felipe,X),born(letizia,Y),later(Y,X).`
- Find two people that were born in the same year
  `?- born(X,f(_,_,Y)),born(Z,f(_,_,Y)),X\=Z.`
- Which is Sofia's birthday? `?- birthday(sofia,X).`

Adding functions

- Note that, in principle, functions are not evaluated. They are just a way to build data structures.

Adding functions

- Note that, in principle, functions are not evaluated. They are just a way to build data structures.
- We usually call them functors, and they are identified by their name and arity (number of arguments). In the example: `f/3`, `d/2`.

Adding functions

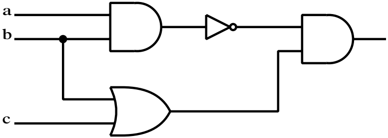
- Note that, in principle, functions are not evaluated. They are just a way to build data structures.
- We usually call them functors, and they are identified by their name and arity (number of arguments). In the example: `f/3`, `d/2`.
- We can use the same name for functors with different arity. For instance, we could have written:
  `birthday(X,date(D,M)) :- born(X,date(D,M,_)).`

Adding functions

- Note that, in principle, functions are not evaluated. They are just a way to build data structures.
- We usually call them functors, and they are identified by their name and arity (number of arguments). In the example: `f/3`, `d/2`.
- We can use the same name for functors with different arity. For instance, we could have written:
  `birthday(X,date(D,M)) :- born(X,date(D,M,_)).`
- As in First Order Logic, we call terms to any combination of functions, constants and variables. In fact, a constant `c` is a 0-ary functor `c/0`.

Adding functions

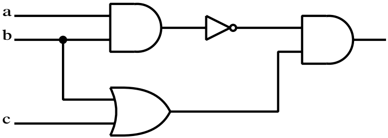
- Example: we can represent a digital circuit.




```
and(not(and(a,b)),or(b,c))
```

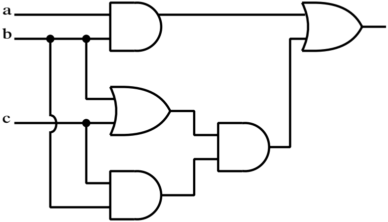
Adding functions

- Example: we can represent a digital circuit.



`and(not(and(a,b)),or(b,c))`

- Exercise: try to represent this circuit



- Arithmetic operators are also (infix) functors. The term `2+3*4` is not equal to `4*3+2` or `14`.

User-defined functors

- We can also define our own functors using the `op` directive.

  ```
  :- op(X,Y,Z).
  ```

  means we declare operator `Z` with precedence number `X` (higher = less priority) and associativity `Y`.

- Associativity can be:
  - infix operators: `xfx` `xfy` `yfx`
  - prefix operators: `fx` `fy`
  - postfix operators: `xf` `yf`

  where:
  - `f`: is the functor position
  - `x`: argument of strictly lower precedence
  - `y`: argument of lower or equal precedence

User-defined functors

- For instance, the fact:

```
equivalent(not(and(A,B)), or(not(A),not(B))).
```

can be written in a more readable way:

```
:- op(800,xfx,<==>).
:- op(700,xfy,v).
:- op(600,xfy,&).
:- op(500,fy,not).
not (A & B) <==> not A v not B.
```

User-defined functors

- For instance, the fact:

  ```
  equivalent(not(and(A,B)), or(not(A),not(B))).
  ```

  can be written in a more readable way:

  ```
  :- op(800,xfx,<==>).
  :- op(700,xfy,v).
  :- op(600,xfy,&).
  :- op(500,fy,not).
  not (A & B) <==> not A v not B.
  ```

- Try the following `?- F=(not a v b & c), F=(H v G).`

User-defined functors

- For instance, the fact:

```
equivalent(not(and(A,B)), or(not(A),not(B))).
```

  can be written in a more readable way:

```
:- op(800,xfx,<==>).
:- op(700,xfy,v).
:- op(600,xfy,&).
:- op(500,fy,not).
not (A & B) <==> not A v not B.
```

- Try the following `?- F=(not a v b & c), F=(H v G).`
- Note that `= > < :- ,` are predefined operators. Predicate `current_op/3` shows the currently defined operators.

Exercise 1

Build a predicate `eval/5` *that computes the output of any circuit for 3 variables so that* `eval(A,B,C,Circuit,X)` *returns the output of* `Circuit` *in* `X` *for values* `a=A`, `b=B` *and* `c=C`.
The predicate must also allow returning the models of the circuit (combinations of values that yield a 1).
Try with the two previous circuits.

Examples:

```
?- eval(1,0,0, a & ( not b v c) ,X).
X = 1.

?- eval(A,B,C, a v not b,1).
A = 1, B = 1 ;
A = 0, B = 0 ;
A = 1, B = 0 ;
```

Unification

- How are functors handled in the goal satisfaction algorithm?

Unification

- How are functors handled in the goal satisfaction algorithm? When searching a goal, we see whether it matches a rule head.

Unification

- How are functors handled in the goal satisfaction algorithm? When searching a goal, we see whether it matches a rule head.
- To see how it works, we can use the built in `=/2` Prolog predicate. Try the following:

```
?- f(X,b)=f(a,Y).
?- f(X,b)=f(X,Y).
?- f(f(Y),b)=f(X,Y).
?- f(f(Y),b)=f(a,Y).
```

Unification

- The general algorithm is well-known: Most General Unifier (MGU) [Robinson 1971].
- Given a set of expressions E, we compute a disagreement set searching from left to right the first different symbol and taking the corresponding subexpression.

Unification

- The general algorithm is well-known: Most General Unifier (MGU) [Robinson 1971].
- Given a set of expressions E, we compute a disagreement set searching from left to right the first different symbol and taking the corresponding subexpression.
- For instance, given $p(f(X), Y)$ and $p(f(g(a, Z), f(Z))$ we get the disagreement set $\{X, g(a, Z)\}$.

Unification

- If two atoms can be unified, they have an MGU that can be computed as follows:

$\sigma := [\,]$;
while $|E| > 1$ {
  $D :=$ disagreement set of E;
  if D contains an X and a term t not containing X {
    $E := E[X/t]$;
    $\sigma := \sigma \cdot [X/t]$; }
  else return 'not unifiable';
}

Unification

- If two atoms can be unified, they have an MGU that can be computed as follows:

$\sigma := [\,];$
while $|E| > 1$ {
  $D :=$ disagreement set of E;
  if D contains an X and a term t not containing X {
    $E := E[X/t];$
    $\sigma := \sigma \cdot [X/t];$ }
  else return 'not unifiable';
}

- Example $E = \{f(f(Y), b), f(X, Y)\}$. Then $D = \{f(Y), X\}$ and we can replace X by $f(Y)$. E becomes $\{f(f(Y), b), f(f(Y), Y)\}$.
- The new disagreement is $D = \{b, Y\}$. After replacing $E[Y/b] = \{f(f(b), b)\}$ and the algorithm stops $\sigma = [X/f(Y)][Y/b]$.

Lists

- A list can be easily implemented with a functor. Take `list(X,L)` where `X` is the head and `L` is the tail. We could use `null` to represent an empty list.

Lists

- A list can be easily implemented with a functor. Take `list(X,L)` where `X` is the head and `L` is the tail. We could use `null` to represent an empty list.
- This is not very readable: `1,2,3,4` would be represented as

```
list(1,list(2,list(3,list(4,null))))
```

Lists

- A list can be easily implemented with a functor. Take `list(X,L)` where `X` is the head and `L` is the tail. We could use `null` to represent an empty list.
- This is not very readable: `1,2,3,4` would be represented as

```
list(1,list(2,list(3,list(4,null))))
```

- Prolog has a predefined operator `'[|]'/2` and a predefined constant `[]` so that a term like

```
'[|]'(1,'[|]'(2,'[|]'(3,'[|]'(4,[]))))
```

can be simply abbreviated as `[1,2,3,4]`

Lists

- We can also write `'[|]'(X,L)` as `[X | L]`.

Lists

- We can also write `'[|]'(X,L)` as `[X | L]`.
- Similarly, `[X, Y, Z | L]` stands for `[X|[Y|[Z|L]]]`
- And `[X, Y, Z]` stands for `[X,Y,Z|[]]`

Lists

- We can also write `'[|]'(X,L)` as `[X | L]`.
- Similarly, `[X, Y, Z | L]` stands for `[X|[Y|[Z|L]]]`
- And `[X, Y, Z]` stands for `[X,Y,Z|[]]`
- Try the query:

```
?- L=[1,2|[3]], L=[1|[2|[3|[]]]].
```

Lists

- We can also write `'[|]'(X,L)` as `[X | L]`.
- Similarly, `[X, Y, Z | L]` stands for `[X|[Y|[Z|L]]]`
- And `[X, Y, Z]` stands for `[X,Y,Z|[]]`
- Try the query:
  `?- L=[1,2|[3]], L=[1|[2|[3|[]]]].`
- Program predicate `member(X,List)`

Lists

- We can also write `'[|]'(X,L)` as `[X | L]`.
- Similarly, `[X, Y, Z | L]` stands for `[X|[Y|[Z|L]]]`
- And `[X, Y, Z]` stands for `[X,Y,Z|[]]`
- Try the query:
  `?- L=[1,2|[3]], L=[1|[2|[3|[]]]].`
- Program predicate `member(X,List)`

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

Lists

- We can also write `'[|]'(X,L)` as `[X | L]`.
- Similarly, `[X, Y, Z | L]` stands for `[X|[Y|[Z|L]]]`
- And `[X, Y, Z]` stands for `[X,Y,Z|[]]`
- Try the query:
  `?- L=[1,2|[3]], L=[1|[2|[3|[]]]].`
- Program predicate `member(X,List)`

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

- Try these queries:

```
?- member(c,[a,b,c,d,c]).
?- member(X,[a,b,c,d,c]).
?- member(a,X).
```

Lists

- Program predicate `append(L1,L2,L3)`

Lists

- Program predicate `append(L1,L2,L3)`

```
append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
```

Lists

- Program predicate `append(L1,L2,L3)`

```
append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
```

- Try these queries:

```
?- append([a,b],[c,d,e],L).
?- append([a,b],L,[a,b,c,d,e]).
?- append(L1,L2,[a,b,c]).
```

Lists

- Program predicate `append(L1,L2,L3)`

```
append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
```

- Try these queries:

```
?- append([a,b],[c,d,e],L).
?- append([a,b],L,[a,b,c,d,e]).
?- append(L1,L2,[a,b,c]).
```

- Use `append` to find the prefix `P` and suffix `S` of a given element `X` in a list `L`. For instance, with `X=wed` and `L=[sun,mon,tue,wed,thu,fri,sat]`, we should get `P=[sun,mon,tue]` and `S=[thu,fri,sat]`.

Lists

- Program predicate `append(L1,L2,L3)`

```
append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
```

- Try these queries:

```
?- append([a,b],[c,d,e],L).
?- append([a,b],L,[a,b,c,d,e]).
?- append(L1,L2,[a,b,c]).
```

- Use `append` to find the prefix `P` and suffix `S` of a given element `X` in a list `L`. For instance, with `X=wed` and `L=[sun,mon,tue,wed,thu,fri,sat]`, we should get `P=[sun,mon,tue]` and `S=[thu,fri,sat]`.
- In the same list, find the predecessor and successor weekdays to some day `X`.

Lists

Exercise 2

1. *Use* `append/3` *to define the predicate* `sublist(S,L)` *so that* `S` *is a sublist of* `L`.

Lists

Exercise 2

1. *Use* `append/3` *to define the predicate* `sublist(S,L)` *so that* `S` *is a sublist of* `L`*.*
2. *Use* `append/3` *to define the predicate* `insert(X,L,L2)` *so that* `X` *is arbitrarily inserted in* `L` *to produce* `L2`*.*

Lists

Exercise 2

1. *Use* `append/3` *to define the predicate* `sublist(S,L)` *so that* `S` *is a sublist of* `L`*.*
2. *Use* `append/3` *to define the predicate* `insert(X,L,L2)` *so that* `X` *is arbitrarily inserted in* `L` *to produce* `L2`*.*
3. *Use* `append/3` *to define the predicate* `del(X,L,L2)` *so that* `X` *is (arbitrarily) deleted from* `L` *to produce* `L2`*.*

Lists

Exercise 2

1. *Use* `append/3` *to define the predicate* `sublist(S,L)` *so that* `S` *is a sublist of* `L`*.*
2. *Use* `append/3` *to define the predicate* `insert(X,L,L2)` *so that* `X` *is arbitrarily inserted in* `L` *to produce* `L2`*.*
3. *Use* `append/3` *to define the predicate* `del(X,L,L2)` *so that* `X` *is (arbitrarily) deleted from* `L` *to produce* `L2`*.*
4. *Use previous predicates to define* `perm(L,L2)` *so that* `L2` *is an arbitrary permutation of* `L`*.*

Lists

Exercise 2

1. *Use* `append/3` *to define the predicate* `sublist(S,L)` *so that* `S` *is a sublist of* `L`*.*
2. *Use* `append/3` *to define the predicate* `insert(X,L,L2)` *so that* `X` *is arbitrarily inserted in* `L` *to produce* `L2`*.*
3. *Use* `append/3` *to define the predicate* `del(X,L,L2)` *so that* `X` *is (arbitrarily) deleted from* `L` *to produce* `L2`*.*
4. *Use previous predicates to define* `perm(L,L2)` *so that* `L2` *is an arbitrary permutation of* `L`*.*
5. *Define predicate* `flatten(L1,L2)` *that removes nested lists putting all constants at a same level in a single list. Example:*

```
?- flatten([[a,b],[c,[d]]],L2).
L2 = [a,b,c,d]
```

The cut predicate

- The cut predicate written `!` behaves as follows:

$$H :- \ B_1, \ldots, B_n, !, B_{n+1}, \ldots, B_m.$$

When `!` is reached, it succeeds but ignores any remaining choice for $B_1, \ldots, B_n$.

The cut predicate

- The cut predicate written `!` behaves as follows:

$$H :-\ B_1, \ldots, B_n, !, B_{n+1}, \ldots, B_m.$$

When `!` is reached, it succeeds but ignores any remaining choice for $B_1, \ldots, B_n$.

- Example: the program

```
max(X,Y,X) :- X>=Y.
max(X,Y,Y) :- X<Y.
```

can be replaced by

```
max(X,Y,X) :- X>=Y,!.
max(X,Y,Y).
```

The cut predicate

- The cut predicate written `!` behaves as follows:

$$H \text{:-}\ \ B_1, \ldots, B_n, !, B_{n+1}, \ldots, B_m.$$

When `!` is reached, it succeeds but ignores any remaining choice for $B_1, \ldots, B_n$.

- Example: the program

```
max(X,Y,X) :- X>=Y.
max(X,Y,Y) :- X<Y.
```

can be replaced by

```
max(X,Y,X) :- X>=Y,!.
max(X,Y,Y).
```

assuming that it is called with an unbounded third variable. Otherwise, a query `max(3,1,1)` will succeed.

The cut predicate

- This second alternative overcomes that problem

```
max(X,Y,M) :-
    X>=Y,!,M=X
  ;  M=Y.
```

The cut predicate

- This second alternative overcomes that problem

```
max(X,Y,M) :-
    X>=Y,!,M=X
  ;  M=Y.
```

- Another example:

```
p(1).
p(2) :- !.
p(3).
```

try the queries

```
?- p(X).
?- p(X),p(Y).
?- p(3).
?- p(X),!,p(Y).
```

The cut predicate

- Typically, it improves efficiency but changes the ways in which a predicate can be used.

The cut predicate

- Typically, it improves efficiency but changes the ways in which a predicate can be used.
- In some cases, it is really necessary for a reasonable solution to a programming problem.

The cut predicate

- Typically, it improves efficiency but changes the ways in which a predicate can be used.
- In some cases, it is really necessary for a reasonable solution to a programming problem. Example: add a non-existing element as head of a list. If existing, leave the list untouched.

The cut predicate

- Typically, it improves efficiency but changes the ways in which a predicate can be used.
- In some cases, it is really necessary for a reasonable solution to a programming problem. Example: add a non-existing element as head of a list. If existing, leave the list untouched.

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

Negation as failure

- The `fail` predicate always fails. The `true` predicate always succeeds.

Negation as failure

- The `fail` predicate always fails. The `true` predicate always succeeds.
- Negation as failure `\+` can be defined as:

```
(\+ P) :- P,!,fail
        ; true.
```

Negation as failure

- The `fail` predicate always fails. The `true` predicate always succeeds.
- Negation as failure `\+` can be defined as:

```
(\+ P) :- P,!,fail
        ; true.
```

- Example: all birds fly, excepting penguins.

```
bird(a).  bird(b).  bird(c).  penguin(b).

fly(X) :- bird(X), \+ penguin(X).
```

Negation as failure

- The `fail` predicate always fails. The `true` predicate always succeeds.
- Negation as failure `\+` can be defined as:

```
(\+ P) :- P,!,fail
        ; true.
```

- Example: all birds fly, excepting penguins.

```
bird(a).  bird(b).  bird(c).  penguin(b).

fly(X) :- bird(X), \+ penguin(X).
```

- Floundering problem: be careful with unbound variables inside negation. The query `?- fly(X).` will fail if using rule `fly(X) :- \+ penguin(X), bird(X).`

Predicate `repeat`

- Predicate `repeat` always succeeds (like `true`) but provides an infinite number of choice points.

Predicate `repeat`

- Predicate `repeat` always succeeds (like `true`) but provides an infinite number of choice points.
- This means that anything that fails afterwards, will return to `repeat` forever.

Predicate `repeat`

- Predicate `repeat` always succeeds (like `true`) but provides an infinite number of choice points.
- This means that anything that fails afterwards, will return to `repeat` forever.
- Its effect can only be canceled by a cut `!`

```
writelist(L) :-
  repeat, (member(X,L), write(X), fail; !).
```

Arithmetics

- Predicate `is` evaluates an arithmetic expression. We can use:
  `+` `-` `*` `/` `**` (power) `//` (integer division) `mod` (modulo).

Arithmetics

- Predicate `is` evaluates an arithmetic expression. We can use:
  `+` `-` `*` `/` `**` (power) `//` (integer division) `mod` (modulo).
- We can make comparisons of numeric values using:
  `>` `<` `>=` `=<` `=:=` `=\=`

Arithmetics

- Predicate `is` evaluates an arithmetic expression. We can use:
  `+` `-` `*` `/` `**` (power) `//` (integer division) `mod` (modulo).
- We can make comparisons of numeric values using:
  `>` `<` `>=` `=<` `=:=` `=\=`
- Examples:

```
gcd(X,X,X) :- !.
gcd(X,Y,D) :- X>Y,!,X1 is X-Y,gcd(X1,Y,D).
gcd(X,Y,D) :- X<Y,gcd(Y,X,D).

length([],0).
length([_|L],N):-length(L,M),N is M+1.
```

Arithmetics

Exercise 3

Define predicate `set_nth0(N,L1,X,L2)` *so that the element of list* `L1` *at position* `N` *(starting from 0) is replaced by* `X` *to produce list* `L2`*.*

Example:

```
?- set_nth0(3,[a,b,c,d,e,f],z,L2).
L2=[a,b,c,z,e,f].
```

Arithmetics

Exercise 4

We have a list of 9 elements that capture the content of a 3 $\times$ 3 grid. The positions in the list corresponds to the grid positions:

0	1	2
3	4	5
6	7	8

Define predicate `nextpos(X,D,Y)` *, so that* `Y` *is the adjacent position to* `X` *following direction* `D` *varying in* `{u,d,l,r}`.

Example:

```
?- nextpos(4,u,X).
X=1.
?- nextpos(4,l,X).
X=3.
```

Input/output

- `write(X)` writes a term on the standard output; `tab(N)` writes `N` spaces; `nl` writes a newline character.

Input/output

- `write(X)` writes a term on the standard output; `tab(N)` writes `N` spaces; `nl` writes a newline character.
- Reading a term from standard input `read(X)`. When the end of file is reached, `X` becomes the special term `end_of_file`.

Input/output

- `write(X)` writes a term on the standard output; `tab(N)` writes `N` spaces; `nl` writes a newline character.
- Reading a term from standard input `read(X)`. When the end of file is reached, `X` becomes the special term `end_of_file`.
- `see(Filename)` changes standard input to `Filename`. When finished, we invoke predicate `seen`.

Input/output

- `write(X)` writes a term on the standard output; `tab(N)` writes `N` spaces; `nl` writes a newline character.
- Reading a term from standard input `read(X)`. When the end of file is reached, `X` becomes the special term `end_of_file`.
- `see(Filename)` changes standard input to `Filename`. When finished, we invoke predicate `seen`.
- Similarly, `tell(Filename)` changes standard output to `Filename`. When finished, we invoke predicate `told`.

Input/output

- `write(X)` writes a term on the standard output; `tab(N)` writes `N` spaces; `nl` writes a newline character.
- Reading a term from standard input `read(X)`. When the end of file is reached, `X` becomes the special term `end_of_file`.
- `see(Filename)` changes standard input to `Filename`. When finished, we invoke predicate `seen`.
- Similarly, `tell(Filename)` changes standard output to `Filename`. When finished, we invoke predicate `told`.
- `put(C)` puts character with code `C` in the standard output.

Input/output

- `write(X)` writes a term on the standard output; `tab(N)` writes `N` spaces; `nl` writes a newline character.
- Reading a term from standard input `read(X)`. When the end of file is reached, `X` becomes the special term `end_of_file`.
- `see(Filename)` changes standard input to `Filename`. When finished, we invoke predicate `seen`.
- Similarly, `tell(Filename)` changes standard output to `Filename`. When finished, we invoke predicate `told`.
- `put(C)` puts character with code `C` in the standard output.
- `get0(C)` gets a character code from standard input. `get(C)` is similar but ignoring blank or non-printable characters.

Assert/retract

- We can modify the database of facts and rules in a dynamic way.
  - `assert(T)` includes new fact/rule `T`.
  - `asserta(T)` includes new fact/rule `T` in the beginning.
  - `assertz(T)` includes new fact/rule `T` in the end.
  - `retract(T)` retracts fact/rule `T`. It fails when not possible (the fact did not match to any existing one).
  - `retractall(T)` like `retract` but retracts all matching facts or rules.

Assert/retract

- We can modify the database of facts and rules in a dynamic way.
  - `assert(T)` includes new fact/rule `T`.
  - `asserta(T)` includes new fact/rule `T` in the beginning.
  - `assertz(T)` includes new fact/rule `T` in the end.
  - `retract(T)` retracts fact/rule `T`. It fails when not possible (the fact did not match to any existing one).
  - `retractall(T)` like `retract` but retracts all matching facts or rules.

- Some Prolog implementations require that predicates are declared as dynamic.

```
:- dynamic user/1.
user(1).
user(2).

?- asserta(user(0)).
?- user(X).
```

Assert/retract

We can use assert/retract to create a “global variable”

```
:- dynamic mycounter/1.

mycounter(0).

increment(X) :-
    retract(mycounter(C)),
    D is C+X,
    assert(mycounter(D)).

?- mycounter(C).
C=0.
?- increment(5), mycounter(C), increment(10).
C=5.
?- mycounter(C).
C=15.
```

Testing the type of terms

- `var(X)` true when `X` is an uninstantiated variable
- `nonvar(X)` true when `X` is not a variable or is already instantiated
- `atom(X)` true when `X` is a symbolic atom
- `integer(X)` true when `X` is an integer number
- `float(X)` true when `X` is a floating point number
- `number(X)` true when `X` is a numeric atom (either integer or float)
- `atomic(X)` true when `X` is atomic (either atom or number)

Dealing with atoms and strings

- Symbolic atoms can contain special characters by using simple quote: `mother('Juana la Loca','Carlos I').`

Dealing with atoms and strings

- Symbolic atoms can contain special characters by using simple quote: `mother('Juana la Loca','Carlos I').`
- The use of double quotes `"Carlos I"` stands for a list of ASCII codes `[67, 97, 114, 108, 111, 115, 32, 73]`.

Dealing with atoms and strings

- Symbolic atoms can contain special characters by using simple quote: `mother('Juana la Loca','Carlos I').`
- The use of double quotes `"Carlos I"` stands for a list of ASCII codes `[67, 97, 114, 108, 111, 115, 32, 73]`.
- `name(A,L)` transforms atom `A` into a list of ASCII codes or vice versa. Examples:

```
?- name('Carlos I',L).

L = [67, 97, 114, 108, 111, 115, 32, 73]

?- append("Hello ","World !",L),name(A,L).

L = [72, 101, 108, 108, 111, 32, 87, 111,
114|...],
A = 'Hello World !'
```

Dealing with atoms and strings

- Any ASCII code for a character *c* can be retrieved by using `0'`*c*. For instance:

```
?- name(A,[ 0'a,0'$,0'.,0'[ ]).

A = 'a$.['
```

Dealing with atoms and strings

- Any ASCII code for a character *c* can be retrieved by using `0'`*c*. For instance:

  ```
  ?- name(A,[ 0'a,0'$,0'.,0'[ ]).

  A = 'a$.['
  ```

- `concat_atom(L,A)` concatenates a list of atoms into a new atom. Example:

  ```
  ?- concat_atom(['Hello ','World ','!'],A).

  A = 'Hello World !'
  ```

Building terms

- The special equiality predicate `X =..L` unifies term `X` with a list `L=[F,A1,A2,...]` where `F` is the main functor of `X` and and `A1,A2,...` its arguments.

Building terms

- The special equiality predicate `X =..L` unifies term `X` with a list `L=[F,A1,A2,...]` where `F` is the main functor of `X` and and `A1,A2,...` its arguments. Examples

```
?- f(a,b) =..  L.

L = [f, a, b]
```

Building terms

- The special equiality predicate `X =..L` unifies term `X` with a list `L=[F,A1,A2,...]` where `F` is the main functor of `X` and and `A1,A2,...` its arguments. Examples

```
?- f(a,b) =..  L.

L = [f, a, b]

?- T=..[+,3,4].

T = 3+4
```

Building terms

- The special equiality predicate `X =..L` unifies term `X` with a list `L=[F,A1,A2,...]` where `F` is the main functor of `X` and and `A1,A2,...` its arguments. Examples

```
?- f(a,b) =..  L.

L = [f, a, b]

?- T=..[+,3,4].

T = 3+4
```

- Process a list of terms so that the numeric arguments of unary functors are increased in one.

```
process([],[]):-!.

process([X|Xs],[Y|Ys]):-
  X =..  [F,A], number(A),!, A1 is A+1,
  Y=..[F,A1], process(Xs,Ys).
process([X|Xs],[X|Ys]):- process(Xs,Ys).
```

Higher order predicates

- Predicate call allows calling other predicates handled as arguments.

Higher order predicates

- Predicate call allows calling other predicates handled as arguments.
- Example: apply some function to a list of numbers

```
double(X,Y) :- Y is 2*X.
minus(X,Y) :- Y is -X.
map([],_,[]).
map([X|Xs],P,[Y|Ys]) :- call(P,X,Y), map(Xs,P,Ys).

?- map([1,3,6],double,L).
?- map([1,3,6],minus,L).
```

Higher order predicates

- Predicate call allows calling other predicates handled as arguments.
- Example: apply some function to a list of numbers

```
double(X,Y) :- Y is 2*X.
minus(X,Y) :- Y is -X.
map([],_,[]).
map([X|Xs],P,[Y|Ys]) :- call(P,X,Y), map(Xs,P,Ys).

?- map([1,3,6],double,L).
?- map([1,3,6],minus,L).
```

- We can also use `=..` to build the term to be called:

```
map([],_,[]).
map([X|Xs],P,[Y|Ys]) :-
    T=..[P,X,Y], T, map(Xs,P,Ys).
```

Higher order predicates

- Predicate `findall(T,G,L)` collects in list `L` all the instantiations for term `T` that satisfy goal `G`

Higher order predicates

- Predicate `findall(T,G,L)` collects in list `L` all the instantiations for term `T` that satisfy goal `G`
- Get a list with all the ancestors of leonor

```
?- findall( X, ancestor(X,leonor), L).
```

- Example: convert a list of elements `[a,b,c,d]` into a list of duplicated pairs

```
?- findall( (X,X), member(X,[a,b,c,d]), L).
```