# Knowledge Representation & Reasoning

**Antonis Troumpoukis**
NCSR “Demokritos”

**Angelos Charalambidis**
NCSR “Demokritos”

**Stasinos Konstantopoulos**
NCSR “Demokritos”

*November 7, 2019*

## Overview of the Course

1. Logic Programming
2. Logic Programming for AI
3. Negation in Logic Programming
4. Answer Set Programming
5. Constraint Satisfaction
6. Higher-order Logic Programming
7. Preference Representation
8. Knowledge Representation in the Semantic Web
9. Reasoning on the Semantic Web
10. Inductive Logic Programming

## Logic Programming

- Prolog (“PROgramming in LOGic”) is the most successful example of the family of logic programming languages.
- A Prolog program is a subset of first-order logic
  - Statements (facts, rules, queries)
  - Terms (data structures)
- Prolog is **declarative**: a Prolog programmer concentrates on **what** the program needs to do, not on **how** to do it.
- Many applications in Artificial Intelligence (use in expert systems, search problems etc.)

## Facts

- are the simplest kinds of statements
- are means of stating that a relation holds between objects

Example

```
parent(john,mary).
```

- John is a **parent** of Mary.
- The relation `parent` holds between the **constants** `john` and `mary`.

- Relations like `parent` are also named as **predicates**.
- `parent` is a binary predicate (takes two **arguments**).

## Facts (cont.)

- representation of a **family tree** using Prolog facts

Example

```
parent(john,mary).
parent(john,anne).
parent(anne,mike).
parent(liza,jack).
male(john).
male(mike).
male(jack).
female(mary).
female(anne).
female(liza).
```

## Queries

- are means of retrieving information from a logic program
- ask whether a certain relation holds between objects

Example

```
?- parent(john,mary).
```

- Is John a **parent** of Mary?
- Does the relation `parent` holds between the constants `john` and `mary`?

## Queries (cont.)

- example queries on a family tree

Example (Input program)

```
parent(john,mary).
parent(john,anne).
parent(anne,mike).
parent(liza,jack).
male(john).
male(mike).
male(jack).
female(mary).
female(anne).
female(liza).
```

Example (Queries)

```
?- parent(john,mary).
true

?- parent(mary,anne).
false
```

## Queries with variables

- Prolog allows the use of **variables**.
- A variable stands for an unspecified individual.
- Variable names always begin with a capital letter.

Example

```
?- parent(X,mary).
```

- Who is the **parent** of Mary?
- Does there exist an individual that is a `parent` of `mary`?

- If the query succeeds, Prolog's execution mechanism will return the corresponding answers.

## Queries with variables (cont.)

Example (Input program)

```
parent(john,mary).
parent(john,anne).
parent(anne,mike).
parent(liza,jack).
male(john).
male(mike).
male(jack).
female(mary).
female(anne).
female(liza).
```

Example (Queries)

```
?- parent(X,mary).
X = john.

?- parent(john,X).
X = mary ;
X = anne.

?- parent(mary,X).
false.
```

## Complex queries

- A complex query consists of several subqueries.

Example

```
?- parent(X,mary), parent(X,anne).
```

- Does there exist an individual `X` that is both a `parent` of `mary` and a `parent` of `anne`?

- the symbol `,` reads as “and”.

Example

```
?- parent(X,Z), parent(Z,mike).
```

- “Who is the **grandparent** of Mike?”

## Rules

- are means of defining new relations in terms of existing relations.

Example

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

- `X` is a `grandparent` of `Y` if there exists some `Z` such that `X` is a `parent` of `Z` and `Z` is a `parent` of `Y`.

- the symbol `:-` reads as “if” and the symbol `,` reads as “and”.

Example

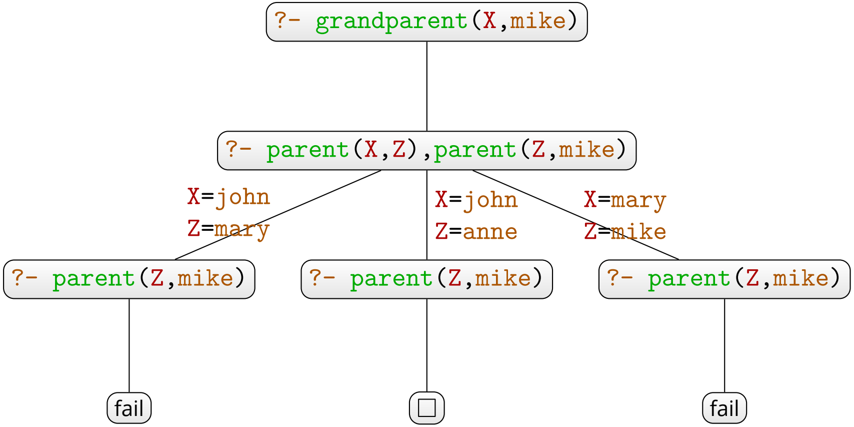
```
?- grandparent(X,mike).
```

- Who is the **grandparent** of Mike?

## Prolog’s execution mechanism

```
parent(john,mary).
parent(john,anne).
parent(anne,mike).

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



## Rules (cont.)

- examples of family members relations

Example

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

son(X,Y) :- parent(Y,X), male(X).
daughter(X,Y) :- parent(Y,X), female(X).

child(X,Y) :- son(X,Y).
child(X,Y) :- daughter(X,Y).
```

## Rules (cont.)

- How about an ancestor relation?

Example

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,A), parent(A,Y).
ancestor(X,Y) :- parent(X,A), parent(A,B), parent(B,Y).
ancestor(X,Y) :- parent(X,A), parent(A,B), parent(B,C), parent(C,Y).
...
```

- This approach is not elegant and
- it would require an *infinite* number of rules to define ancestor relation correctly.

## Recursive Rules

- Prolog allows the definition of relations that are defined in terms of themselves.

Example

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

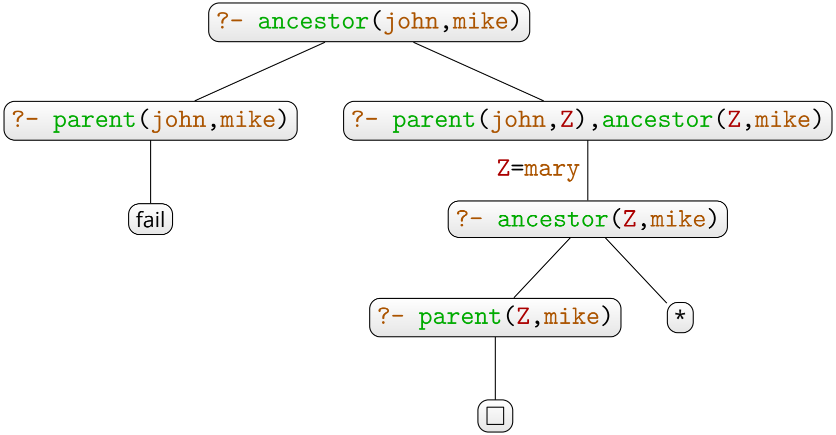
- `X` is an `ancestor` of `Y`
  1. if `X` is a `parent` of `Y`, or
  2. if there exists some `Z` such that
     `X` is a `parent` of `Z` and `Z` is an `ancestor` of `Y`.

- The second rule is recursive; it uses ancestor to define ancestor.
- The first rule stops the predicate from calling itself.

# Prolog's execution mechanism (cont.)

```
parent(john,anne).
parent(anne,mike).

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



* failing proof tree

## Terms

- How to define a `plus` relation using constants?

Example

```
plus(zero,zero,zero).
plus(one,zero,one).
plus(zero,one,one).
plus(one,one,two).
plus(two,zero,two).
plus(zero,two,two).
...
```

- This approach would require an *infinite* number of constants and facts.

## Terms (cont.)

- A **term** can be one of the following:
  1. a **constant**
  2. a **variable**
  3. a $n$-ary **functional symbol** applied to $n$ terms.

Example

- Representation of natural numbers using compound terms:
  1. the constant `0` which denotes the number $0$
  2. a unary functional symbol `s` which denotes the **successor** function.

$$0 \equiv \texttt{0}, \quad 1 \equiv \texttt{s(0)}, \quad 2 \equiv \texttt{s(s(0))}, \quad 3 \equiv \texttt{s(s(s(0)))}, \quad \ldots$$

## Natural numbers and compound terms

- Implementation of `plus` using compound terms

Example

```
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

- $0 + x = x$
- $\mathbf{s}(x) + y = \mathbf{s}(x + y)$, where $\mathbf{s}(n) = n + 1$.
- `plus(X,Y,Z)`: `X`, `Y` and `Z` are natural numbers such that `Z` is the **sum** of `X` and `Y`.

Example

```
?- plus(s(0),s(0),K).
K = s(s(0))
```

## Natural numbers and compound terms (cont.)

- Multiple uses can be made for a single program

Example

```
?- plus(s(0),s(0),s(s(0))).
```

- Confirm that $1 + 1$ equals $2$.

Example

```
?- plus(s(0),X,s(s(s(0)))).
```

- Calculate the difference $3 - 1$.

Example

```
?- plus(s(0),s(s(0)),X)).
```

- Calculate the sum $1 + 2$.

Example

```
?- plus(X,Y,s(s(s(0)))).
```

- Find all pairs that add up to $3$.

## Natural numbers and compound terms (cont.)

Example

```
times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,W), plus(W,Y,Z).
```

- $0 \cdot x = 0$
- $\mathbf{s}(x) \cdot y = (x + 1) \cdot y = x \cdot y + y.$
- `times(X,Y,Z)`: `X`, `Y` and `Z` are natural numbers such that `Z` is the **product** of `X` and `Y`.

- Using this representation we can define additional numeric operators, such as `power`, `factorial`, etc.
- This approach is not so efficient, used an introductory example to compound terms.

## Arithmetic operators in Prolog

- Prolog supports **integers** and **real** numbers
- Prolog offers built-in predicates for numerical **comparison** (`<`, `=<`, `>`, `>=`, etc.)
- Prolog offers built-in predicates for arithmetic **operations** (`+`, `−`, `*`, `/`, etc.)
  - These operators are actually functional symbols:
    e.g., `1+2` corresponds to `+(1,2)` and not to `3` as in other programming languages
- We can “force” Prolog to evaluate things using the built-in binary operator `is`
  - Usage: `V is E` succeeds if `V` is equal to the number that `E` evaluates.

## Arithmetic operators in Prolog (cont.)

Example

```
?- 8 is 5+3.  % Evaluate 5+3 and confirm that this is equal to 8
true.

?- 5+3 is 8.  % +(5,3) is different from 8
false.

?- X is 5+3.  % Evaluate 5+3 and place it in variable X
X = 8.

?- 8 is 5+Y.  % Cannot evaluate 5+Y
ERROR: is/2: Arguments are not sufficiently instantiated
```

## Compound terms and data structures

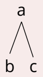
**Example**

- Representation of lists using compound terms:
  1. the constant `nil` which denotes the **empty list**
  2. a binary functional symbol `list` used to denote the **head** and the **tail** of the given list.

$\boxed{a} \equiv$ `list(a,nil)`, $\begin{array}{|c|c|c|}\hline a & b & c \\ \hline\end{array} \equiv$ `list(a,list(b,list(c,nil)))`

**Example**

- Representation of binary trees using compound terms:
  1. the constant `void` which denotes the **empty tree**
  2. a ternary functional symbol `tree` used to denote the **root** of the tree, the left and the right **subtrees** of the given tree.



$\equiv$ `tree(a,tree(b,void,void),tree(c,void,void))`

## Lists in Prolog

- The **empty list** is represented as `[]`
- Non-empty lists: e.g., $\begin{array}{|c|c|c|}\hline a & b & c \\ \hline\end{array}$ is represented as `[a,b,c]`
- The operator `|` shows the **head** and the **tail** of the list.
  - `[X|L]` denotes that `X` is the head of the list and `L` is the tail of the list.
  - `[a,b,c]` can be written as `[a|[b,c]]` and
    `[a]` can be written as `[a|[]]`
  - Used for **pattern matching**:
    `[a|L]` matches any list that begins with `a` and
    `[a,b|L]` matches any list that begins with `a,b`

## Lists in Prolog (cont.)

- Membership of a list

Example

```
member(X,[X|Xs]).
member(X,[Y|Xs]) :- member(X,Xs).
```

- `X` is a `member` of `L`
  1. if `X` is the head of `L`, or
  2. `X` is a `member` of the tail of `L`.

Example

```
?- member(b,[a,b]).
true
```

Example

```
?- member(X,[a,b]).
X = a;
X = b
```

## Lists in Prolog (cont.)

- List concatenation.

Example

```
append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

- the concatenation of an empty list and a list `L` is the list `L`
- the concatenation of an non-empty list `L` with a list `M` is a list which has as head the head of `L` and as tail the concatenation of the tail of `L` and `M`.

- The structure of `append` resembles the structure of `plus`.

## Lists in Prolog (cont.)

- Uses of `append`:

Example

```
?- append([a,b],[c,d],L).
L = [a, b, c, d].
```

Example

```
?- append([a,b],L,[a,b,c,d]).
L = [c, d].
```

Example

```
?- append(X,Y,[a,b,c]).
X = [],
Y = [a, b, c] ;
X = [a],
Y = [b, c] ;
X = [a, b],
Y = [c] ;
X = [a, b, c],
Y = []
```

## Lists in Prolog (cont.)

Example

```
prefix(P,L) :- append(P,Z,L).
```

- P is a prefix of L if there exists a list Z such that if we append Z in P we have the list L.

Example

```
reverse([],[]).
reverse([X|Xs],R) :- reverse(Xs,Rs), append(Rs,[X],R).
```

- the reverse of an empty list is the empty list
- the reverse of a non-empty list L can be obtained by appending a list that contains only the head of L at the end of the reverse of the tail of L.

# A classic riddle

- A man travels with **wolf**, **goat** and **cabbage**
- Wants to cross a river from west to east
- A rowboat is available, but only large enough for the **man** plus one possession
- **Wolf** eats **goat** if left alone together
- **Goat** eats **cabbage** if left alone together
- How can the man cross without loss?

## Configurations

- Represent a configuration of this system as a list `[Man,Wolf,Goat,Cabbage]` showing which bank each thing is
- `w` denotes the **west** bank
- `e` denotes the **east** bank
- Initial configuration: `[w,w,w,w]`
- If man crosses with wolf, new state is `[e,e,w,w]`
  but then goat eats cabbage, so we can't go through that state!
- Desired state: `[e,e,e,e]`

## Moves

- In each move, man crosses with at most one of his possessions
- We will represent these four moves with four constants:
  1. `wolf` denotes that the man crosses the river with wolf
  2. `goat` denotes that the man crosses the river with goat
  3. `cabbage` denotes that the man crosses the river with cabbage
  4. `nothing` denotes that the man crosses the river alone
- Each move transforms one configuration to another
  - e.g., `wolf` transforms the `[w,w,w,w]` into `[e,e,w,w]`

## Moves (cont.)

- In Prolog, we will write this as a predicate `move(Config,Move,NextConfig)`
  1. `Config` is a configuration (like `[w,w,w,w]`)
  2. `Move` is a move (like `wolf`)
  3. `NextConfig` is the resulting configuration (in this case, `[e,e,w,w]`

```
change(e,w).
change(w,e).

move([X,X,Goat,Cabb],wolf,[Y,Y,Goat,Cabb]) :- change(X,Y).
move([X,Wolf,X,Cabb],goat,[Y,Wolf,Y,Cabb]) :- change(X,Y).
move([X,Wolf,Goat,X],cabbage,[Y,Wolf,Goat,Y]) :- change(X,Y).
move([X,Wolf,Goat,Cabb],nothing,[Y,Wolf,Goat,Cabb]) :- change(X,Y).
```

## Safe configurations

- A configuration is **safe** if:
  - **wolf** and **goat** are either guarded by man or are in different banks
  - **goat** and **cabbage** are either guarded by man or are in different banks

```
guarded_or_separated(X,X,X).
guarded_or_separated(_,w,e).
guarded_or_separated(_,e,w).

safe([Man,Wolf,Goat,Cabbage]) :-
      guarded_or_separated(Man,Wolf,Goat),
      guarded_or_separated(Man,Goat,Cabbage).
```

## Solutions

- A **solution** for a **starting configuration** is a **list of moves** that takes you to the goal configuration, where all the intermediate configurations are **safe**.
    1. A solution for `[e,e,e,e]` would be the **empty list** (no moves are needed).
    2. Otherwise, a solution for `Config` is defined recursively as **one move** that takes you to a new safe configuration `NextConfig` followed by the **solution** for `NextConfig` .

```
solution([e,e,e,e],[]).
solution(Config,[FirstMove|OtherMoves]) :-
      move(Config,FirstMove,NextConfig),
      safe(NextConfig),
      solution(NextConfig,OtherMoves).
```

## Finding a solution

```
?- solution([w,w,w,w],L).
ERROR: Out of local stack
```

- Nothing in this Prolog program specifies how long the solution has to be.
- It gets lost looking at possible solutions like `[goat,goat,goat,goat,goat,...]`.
- To solve this, we force the lengths of the solution to be of finite length.

## Finding a solution (cont.)

- `length(L,N)`: the length of the list `X` is equal to `N`.

Example

```
?- length(L,N).
L = [],
N = 0 ;
L = [_],
N = 1 ;
L = [_, _],
N = 2 ;
L = [_, _, _],
N = 3 ;
...
```

## Finding a solution (cont.)

```
?- length(L,N), solution([w,w,w,w],L).
L = [goat, nothing, wolf, goat, cabbage, nothing, goat],
N = 7
```

- "First cross with the goat, then cross back empty,
  then cross with the wolf, then cross back with the goat,
  then cross with the cabbage, then cross back empty,
  and finally cross with the goat".
- this solution has the minimum number of steps.