Prolog Examples

Distributed Systems / Technologies

Sistemi Distribuiti / Tecnologie

Giovanni Ciatto
giovanni.ciatto@unibo.it

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018

What We Will Learn

1. *N*-Queens
   - Thinking about Goals
   - Iteration & Return Values
   - The *cut* Operator
   - One-way Predicates
   - The Power of Backtracking
   - Partially-defined Structures
2. Tic Tac Toe
   - Knowledge Representation
   - Dealing with Side-Effects
   - Prolog and AI
   - Reasoning about (Logic) space
   - CWA Implications
   - Meta-Programming

Prolog Example #1: N-Queens

Brief description of the N-Queens game

We want to dispose N queens[a] on a $N \times N$ chessboard in such a way that no queen is able to attack the other ones.

[a] A piece from the well known "Chess" strategy game. A queen can attack any other piece being on the same row, column, diagonal and anti-diagonal

A solution to the 8-Queens problem

A solution to the N-Queens problem is commonly represented as a N-uple of column-row (or X-Y) coordinates representing the queens' positions on the chessboard. For instance:

$$(1,4),\ (2,2),\ (3,7),\ (4,3),\ (5,6),\ (6,8),\ (7,5),\ (8,1)$$

Informal Decomposition of the Problem

- we expect the solution to be a *list* of coordinates representing queens positions on the board:
  - e.g., `[(X1, X2), ..., (XN, YN)]`
  - we need a `n_queens(+N:int, -Cells:list)` predicate, iterating over *columns* and assigning *rows* to queens
- we need to test whether some queen in position `(X, Y)` may *attack* another queen in position `(X1, Y1)` or not
  - we need a `noattack(+Cell1, +Cell2)` predicate, checking if the first queen *is not* able to attack the second one
  - we may actually want to test this property against more than two queens...
- it may be useful to *range* over numbers from 1 to N
  - we need a `range(+Fst:int, +Lst:int, -Range:list)` predicate, *generating* the list of numbers `[Fst, Fst + 1, ..., Lst]`

The `range/3` Predicate – Try it Yourself

Advices for `range(+Fst:int, +Lst:int, -Range:list)`

- in the general case (`Fst < Lst`), the following conditions should hold:
  - the head of `Range` should unify with `Fst`
  - the head of the tail of `Nums` should unify with `Fst + 1`
  - the head of the tail of the tail of `Nums` should unify with `Fst + 2`
  - ...
- whenever `Fst = Lst`, the returning list should unify with the singleton `[Lst]`.

`range/3` usage examples

- `?- range(1, 5, L).` $\longrightarrow$ `L/[1, 2, 3, 4, 5].`
- `?- range(-1, 2, [-1, 0, 1, 2]).` $\longrightarrow$ `yes.`
- `?- range(6, 6, L).` $\longrightarrow$ `L/[6].`
- `?- range(7, 6, _).` $\longrightarrow$ `no.`

The range/3 Predicate – Possible Solution

Our proposal

```
% range(+F, +L, -R)
range(L, L, [L]).
range(F, L, [F | O]) :-
  F < L,
  N is F + 1,
  range(N, L, O).

% L -> Last
% F -> First
% O -> Others
% N -> Next
% R -> Range
```

- try querying `?- range(1, 5, L)`
  - why is the engine proposing to search for alternative solutions?
  - is the engine able to provide alternative solutions?
- try replacing the `is`/2 operator with the `'='`/2 and see what changes.
- try removing the `F < L` and see what changes.
- note that "results" are "built" by providing *unbounded* variables to invoked predicates and letting them bind such variables to some partial result

The *cut* Operator: `'!'/0`

Informal definition

The cut operator (`'!'/0`) is a 0-arity atom always evaluating to `true`. Its aim is to provide a powerful *side-effect*. It prevents *backtracking* to any un-explored branch of the proof tree prior to the cut evaluation.

Cut example 1

Try replacing `range(L, L, [L]) :- !.` to the non-recursive case of the `range/3` predicate definition. What changes?

Cut example 2

```
a(1). b(2). c(3).
val(X) :- a(X).
val(X) :- b(X)/*, !*/.
val(X) :- c(X).
```

- try querying `?- val(X)` and keep accepting until no more solution is provided.
- now remove the comments and retry.

The `noattack/2` Predicate – Try it Yourself

Advices for `noattack(+Cell1, +Cell2)`

A queen in position `Cell1 = (X1, Y1)` *cannot attack* another queen in position `Cell2 = (X2, Y2)` if `Cell1` and `Cell2`

- are not on the same *row*
- are not on the same *column*
- are not on the same *diagonal*
- are not on the same *anti-diagonal*

`noattack/2` usage examples

- `?- noattack((1, 1), (2, 3)).` $\longrightarrow$ `yes.`
- `?- noattack((3, 4), (3, 7)).` $\longrightarrow$ `no.`
- `?- noattack((7, 3), (3, 7)).` $\longrightarrow$ `no.`
- `?- noattack((1, 7), (3, 7)).` $\longrightarrow$ `no.`

The `noattack/2` Predicate – Possible Solution

Our proposal

```
noattack((X1, Y1), (X2, Y2)) :-
  X1 =\= X2,  Y1 =\= Y2,    % not same col and not same row
  (Y2 - Y1) =\= (X2 - X1), % not same diagonal
  (Y2 - Y1) =\= (X1 - X2). % not same anti-diagonal
```

- what if the actual arguments are not couples?
- what if the actual arguments are not couples *of numbers*?
- what if the actual arguments contains one or more unbounded variables?
  - e.g. what's the expected outcome of `?- noattack((2,2),(X,Y)).`
  - can this predicate be used to *generate* a non-attacking couple of cells?
  - why?

The `noattack/2` Predicate – Generalisation

Try it yourself

Try now to generalize the `noattack/2` predicate to check the case where a queen in `Cell = (X, Y)` is not able to attack any queen in positions `Cells = [(X1,Y1), (X2,Y2), ...]`

The `noattack/2` Predicate – Generalisation

Try it yourself

Try now to generalize the `noattack/2` predicate to check the case where a queen in `Cell = (X, Y)` is not able to attack any queen in positions `Cells = [(X1,Y1), (X2,Y2), ...]`

Our proposal

```
noattack((X1, Y1), (X2, Y2)) :- !, % which is the purpose
  X1 =\= X2,  Y1 =\= Y2,           % of the cut here?
  (Y2 - Y1) =\= (X2 - X1),
  (Y2 - Y1) =\= (X1 - X2).

noattack(_, []).
noattack(Cell, [HeadCell | Others]) :- % where's recursion?
    noattack(Cell, HeadCell), % which one is called here?
    noattack(Cell, Others).   % which one is called here?
```

The `solution/2` Predicate – Try it Yourself I

Let's assume `Template = [(X1, Y1), ..., (XN, YN)]`, where each `Yi` is *unbound* and all `Xi` are assigned to all values in the set `{1..N}`.

- we will enforce this assumption later

Advices for `solution(+N:int, ?Template:list)`

Under such hypotheses, a solution to the `N`-Queens problem can be computed as follows

1. choose a queen-free column, i.e. choose some `Xi` in `Template` such that `Yi` is still unassigned
2. select some value in `{1..N}` and assign it to `Yi`
3. ensure queen `(Xi, Yi)` is *not* attacking any *already located* one
4. once all `Yi` have been assigned, you're done

The `solution/2` Predicate – Try it Yourself II

`solution/2` usage examples

- `?- solution(1, [(1, Y1)]).` $\longrightarrow$ `Y1/1.`
- `?- solution(2, [(1, Y1), (2, Y2)]).` $\longrightarrow$ `no.`
- `?- solution(3, [(1, Y1), (2, Y2), (3, Y3)]).` $\longrightarrow$ `no.`
- `?- solution(4, [(1, Y1), (2, Y2), (3, Y3), (4, Y4)]).` $\longrightarrow$ `Y1/3, Y2/1, Y3/4, Y4/2.`
- `?- solution(8, [(1, Y1), (2, Y2), (3, Y3), (4, Y4), (5, Y5), (6, Y6), (7, Y7), (8, Y8)]).` $\longrightarrow$ `Y1/4, Y2/2, Y3/7, Y4/3, Y5/6, Y6/8, Y7/5, Y8/1`

The `solution/2` Predicate – Possible Solution I

Our proposal

```
% solution(+N:int, T:list)
solution(_, []).
solution(N, [(X, Y) | T]) :-
  solution(N, T),
  range(1, N, R), % R=[1..N]
  member(Y, R),
  noattack((X, Y), T).

% N -> num. of queens
% T -> template
% R -> range
```

- why is the recursive call appearing as the *first* subgoal of `solution/2`?
  - what if it was the last call?
- where are we choosing a "queen-free" column?
- where are we ensuring the current candidate queen is not attacking any "already located" one?
- isn't `Y` unbounded? Where are we selecting a value for `Y`?
- if the selected value is invalid, where are we selecting another one?

The `solution/2` Predicate – Possible Solution II

Why recursion is on first sub-goal? (informally)

- Prolog search strategy is *depth-first*

→ when iterating over the elements of some lists...

  - if recursion is on *last* subgoal, elements are visited *from left to right*,
  - if recursion is on *first* subgoal, elements are visited *from right to left*.

Try `?- L = [a, b, c, d], lprintall(L), rprintall(L).`

```
lprintall([]).
lprintall([X | Xs]) :-
  write(X), nl,
  lprintall(Xs).
```

```
rprintall([]).
rprintall([X | Xs]) :-
  rprintall(Xs),
  write(X), nl.
```

The `solution/2` Predicate – Possible Solution III

- we are actually visiting columns from right to left
- assigning `Yi` from right to left
- ! so `T` — the tail of the template — always contains the "already located" queens...
  - where "always" means "in any iteration step of `solution/2`"
- ... thus it is sufficient to check that queen in `(X, Y)` is not attacking any queen in `T`.

The `solution/2` Predicate – Possible Solution IV

```
?- member(Y, [1, 2, ..., N]), Goal.
```

Assuming `Y` is initially *unbounded* and `Goal` employs `Y` somehow, this can be read as:

Select a number in $\{1..N\}$, assign it to `Y` and try to prove `Goal`.
On failure, select another number and retry.

The "retry" semantics is implicit because of Prolog's backtracking!

The `template/2` Predicate – Try it Yourself

Advices for `template(+N:int, -Template:list)`

We need this predicate to produce the partially-assigned `Template = [(1, Y1), ..., (N, YN)]` assumed as second argument for the `solution/2` predicate.

- One could think to map each element `X` in the *range* `[1, ..., N]` into a couple `(X, Yi)`, where `Yi` is *unbounded*

`template/2` usage examples

- `?- template(1, L).` $\longrightarrow$ `L = [(1,Y)].`
- `?- template(2, L).` $\longrightarrow$ `L = [(1,Y1), (2,Y2)].`
- `?- template(4, L).` $\longrightarrow$ `L = [(1,Y1), (2,Y2), (3,Y3), (4,Y4)].`

Please note that tuProlog may arbitrarly rename unbounded variables.

The `template/2` Predicate – Possible Solution

Our proposal

```
template(N, Template) :-
  range(1, N, Xs),
  template(Xs, Template).

template([], []).
template([X | Xs], [(X, _) | XYs]) :-
  template(Xs, XYs).
```

- what if we replace the anonymous variable '_' by a named variable Y?
- what kind of template would we obtain?
- is the rules occurrence order relevant?

The important thing to note here is that partially-defined data structures can be easily defined in Prolog and are powerful to use.

The `n_queens/2` Predicate: Closing the Circle

Wiring up

```
n_queens(N, Solution) :-
  template(N, Solution),
  solution(N, Solution).
```

Ultimate tasks

1. try querying `?- n_queens(8, S).`
   - Enjoy ;)
2. now implement a solver for the *N*-Queens problem in Java.

What We Will Learn

1. *N*-Queens
   - Thinking about Goals
   - Iteration & Return Values
   - The *cut* Operator
   - One-way Predicates
   - The Power of Backtracking
   - Partially-defined Structures
2. Tic Tac Toe
   - Knowledge Representation
   - Dealing with Side-Effects
   - Prolog and AI
   - Reasoning about (Logic) space
   - CWA Implications
   - Meta-Programming

Prolog Example 2: Tic-Tac-Toe

Brief description of the Tic-Tac-Toe game

Well known 2-players turn-based discrete-world game. The players must locate their own *symbols* (either `'x'` or `'o'`) over a 3×3 initially empty board. Once a symbol has been assigned to an empty cell, it cannot be removed. The first player *aligning* 3 symbols of his/her is the winner. If the board is *full* and no player has aligned 3 symbols, the match is *even*.

The state of a Tic-Tac-Toe match

The state of a Tic-Tac-Toe match consists of

- the next expected symbol
- the position of the already located symbols

These information can be represented in several ways. Each representation may influence the way one may *reason* about the state of the match.

Problems to be Faced

- several possible representation of the state
  - should we only represent already located symbols?
  - should we represent empty cells to?
  - how to represent empty cells?
  - why do representations differ?
- state will eventually change due to players moves: unlike N-Queens, Tic Tac Toe is not pure computation $\rightarrow$ side effectes should be handled here
  - how to deal with side effects and state changes?
- how can we exploit Prolog to reason about a Tic Tac Toe match in a declarative way?
  - can we build artificial players?
  - how difficult is it?

Representing State – Possible Solution

Our proposal

```
turn(o).

cell(1, 1, o).
cell(1, 2, e).
cell(1, 3, x).
cell(2, 1, e).
cell(2, 2, x).
cell(2, 3, e).
cell(3, 1, e).
cell(3, 2, e).
cell(3, 3, e).

%   o _ _
%   _ x _
%   x _ _
```

- ! the whole board is *explicitly* represented
- ! the "next expected symbol" (i.e. the turn) is represented by means of the `turn/1` predicate
- ! "cross" (`'x'`), "circle" (`'o'`) and "empty" (`'e'`) symbols are represented as constants
- what do we mean by querying `?- cell(X, Y, x).?`
  - what by `?- cell(X, Y, e).?`
  - what by `?- cell(2, 3, S).?`
  - what by `?- cell(X, Y, S).?`
- this is just an example: initially, all cells should be empty

Representing State – Alternative Solutions I

Representing all cells, empty symbol with '_'

e.g. `cell(1, 2, _). cell(2, 1, _). %...`

- how would you check if a particual cell is empty?
- how would you check if a particual cell contains symbol `S`?
- how would you iterate over empty (resp. non-empty) cells?

Representing only non-empty cells

e.g. `cell(1, 1, o). cell(1, 3, x). cell(2, 2, x).`

- same questions as above.
  - same answers?

Representing State – Alternative Solutions II

Is keeping track of the turn really necessary?

We could assume the *initual* turn to always be `'x'`. We could then compute the turn by counting the amount of non-empty cells[a].

- would this computation be affected by the chosen representation of the board?
- how?

[a] even number $\Longrightarrow$ `'x'`, odd number $\Longrightarrow$ `'o'`

Changing State Dynamically I

- logic atoms are immutable
  - `cell(1, 2, e)` is an axiom $\Longrightarrow$ cell `(X, Y)` is empty — now and forever — according to 1OL semantics

$\times$ this is not true in practice since the sentence "cell `(X, Y)` is empty" may eventually turn from false to true after some player move

$\rightarrow$ Prolog allows the program to *dynamically* update its *Knoledge Base* (KB) during its own execution

- KB $\approx$ Theory $+$ Dynamically added facts and rules

- if player 2 succeeds in moving `'o'` to cell `(1, 2)`, then we should
  - *retract* our knowledge about the *fact* `cell(1, 2, e)`
  - *asserting* that `cell(1, 2, o)` instead.

The `assert/1` and `retract/1` Predicates

Adding clauses to the KB

- `assertz(Klause)`, or simply `assert(Klause)`, evaluate to `true`, with the side effect that the clause `Klause` is appended at the end of the KB
- `asserta(Klause)` evaluates to `true`, with the side effect that the clause `Klause` is added to the beginning of the KB

! why do we need to choose where to insert clauses?

Removing clauses from the KB

- `retract(Klause)` evaluates to `true`, with the side effect of removing from the KB a clause that matches `Klause` (which must be at least partially instantiated); multiple solutions are given upon backtracking

The `move/3` Predicate – Try it Yourself I

Advices for `move(+X:int, +Y:int, +S:const)`

This predicate is used by players to make their moves. It moves symbol `S` into `(X, Y)`, after ensuring that

- the next expected symbol is `S`
- cell `(X, Y)` is empty

If the operation succeeds, the next expected symbol is updated accordingly

! you may need to write some support predicate

The `move/3` Predicate – Try it Yourself II

`move/3` use cases

Suppose the KB coincides with the state representation shown in slide 23.

! you should reload your theory after each test in order to revert the side-effects.

- `?- move(2, 2, o).` ⟶ `no.`
- `?- move(1, 1, o).` ⟶ `no.`
- `?- move(3, 1, o).` ⟶ `yes.`
- `?- move(3, 1, o), move(3, 1, o).` ⟶ `no.`
- `?- move(3, 1, o), move(3, 1, o).` ⟶ `no.`
  - (without reverting side effects) `?- move(2, 1, x).` ⟶ `yes.`

The `move/3` Predicate – Possible Solution

Our proposal

```
% move(+X, +Y, +S)
move(X, Y, S) :-
  turn(S),
  retract(cell(X, Y, e)),
  asserta(cell(X, Y, S)),
  next_turn(S, NS),
  retract(turn(S)),
  asserta(turn(NS)),

% next_turn(?S, ?S)
next_turn(x, o).
next_turn(o, x).
```

- where are we checking if `S` is the next expected turn?
- where are we checking if `(X, Y)` is empty?
- try replacing `asserta/1` with `assertz/1`: is it different in this case?
  - is it different in the general case?
- is the order of subgoals in `move/3` relevant, and why?
- what is the purpose of `next_turn/2`?

Drafting the `aligned/2` Predicate I

Suppose `aligned/2` is available, having the following semantics:

`aligned(+Symbols:list, ?Cells:list)`

- `Symbols` is an *input* list of symbols
  - e.g. `[x, x, e]`
- `Cells` is an *output* list of cells
  - e.g. `[cell(1,3,x), cell(2,2,x), cell(3,1,e)]`
- the predicate searches a *sequence of cells* whose symbols match the input sequence of symbols, *regardless of their orientation* on the board

Drafting the `aligned/2` Predicate II

`aligned/2` usage examples

Suppose the KB coincides with the state representation shown in slide 23.

- you should reload your theory after each test in order to revert the side-effects.

?- `aligned([o, e, e], L).` $\longrightarrow$ `L/[cell(1,1,o), cell(2,1,e), cell(3,1,e)]`

?- `aligned([e, e, e], L).` $\longrightarrow$ `L/[cell(3,3,e), cell(3,2,e), cell(3,1,e)] ; L/[cell(3,1,e), cell(3,2,e), cell(3,3,e)]`

?- `aligned([S, S, S], _), S \== e.` $\longrightarrow$ `no.`

Drafting the `aligned/2` Predicate III

Why is `aligned/2` so useful?

- we do not need to reason on coordinates anymore
- we can easily write orientation-independent rules
- we can write the *artificial intelligence* (AI) of a Tic-Tac-Toe player simply combining the `aligned/2` and `move/3` predicates

e.g. let us pretend to be the AI driving player 2 moves with symbol `'o'`

Drafting the `aligned/2` Predicate IV

AI of player 2 ('o')

```
% is the match over?
?- not cell(_, _, e),    % Yes, if there's no empty cell
   aligned([S,S,S], _). % and 3 aligned symbols exist
% if yes and S = o then I'm winning

% if I can win, do it
?- (aligned([o,e,o], [_,cell(X,Y,e),_]); % semicolon = OR
   aligned([o,o,e], [_,_,cell(X,Y,e)])),
   move(X, Y, o).

% if I'm going to loose, avoid it
?- (aligned([x,e,x], [_,cell(X,Y,e),_]); % semicolon = OR
   aligned([x,x,e], [_,_,cell(X,Y,e)])),
   move(X, Y, o).

% ... other rules
```

The `aligned/2` Predicate – Try it Yourself

Advices for the `aligned/2` predicate

- we need to check if some cell is northward to another
  - and if each cell *in a list* is northward to *the next one*
  - ! note that Y-axis progresses up-down
- ...
- we actually need to check this for all cardinal directions
  - i.e. north, south, east, west, northeast, northwest, southeast, southwest
- we need a way to transform a sequence of symbols `[S1, S2, ...]` into a sequence of partially-assigned cells `[cell(X1,Y1,S1), cell(X2,Y2,S2), ...]`
- finally, cells *in a list* are aligned if *there exists a direction* such that all cells are on that direction

The aligned/2 Predicate – Possible Solution I

Some utilities (repeat for each direction)

```
% north(?Cell1:tuple, ?Cell2:tuple)
north(cell(X,Y1,S1), cell(X,Y2,S2)) :-
  cell(X, Y1, S1), % Why this subgoal?
  cell(X, Y2, S2), % Why this subgoal?
  1 is Y2 - Y1.

% north(?Cells:list)
north([C1, C2]) :- north(C1, C2).
north([C1, C2 | Cs]) :-
  north(C1, C2),
  north([C2 | Cs]). % Recursion's here

% line(?Cells:list)
line(Cells) :- north(Cells).

% Other directions are analogous
```

- why there's no need for X1 and X2 in north/2?
- what's the purpose of cell(X,Y1,S1) & cell(X,Y2,S2)?
- why not Y2 - Y1 is 1?
- what's the purpose of north/1?
- what's the purpose of line/1?

The `aligned/2` Predicate – Possible Solution II

From symbols lists to cells lists (and vice versa)

```
% pattern(?Cells, ?Symbols)
pattern([], []).
pattern([cell(_,_,S) | Cs], [S | Ss]) :-
    pattern(Cs, Ss).
```

Wiring up

```
aligned(Symbols, Cells) :-
  pattern(Cells, Symbols),
  line(Cells).
```

- is `pattern/2` generating lists of symbols from lists of cells or vice versa, and why?
- what is the difference if we switch `line` and `pattern` subgoals?

Consequences of the Closed World Assumption I

Closed World Assumption (CWA)

What is not known to be true, is false.

- this is the underlying assumption in Prolog

$\rightarrow$ whatever cannot be inferred from the KB is evalued to `false`.

Consequences of the Closed World Assumption II

Negation as failure (NaF): `not Goal`

Remember subgoal `not cell(_,_,e)` from the AI example?

- it succeeds if the engine fails at proving `cell(_,_,e)`
- ... which requires visiting the whole proof-tree of `?-cell(_,_,e).`
  - it might become quite inefficient

NaF: possible implementation

```
not(Goal) :- call(Goal), !, fail.
not(_).
```

Consequences of the Closed World Assumption III

Not everything can be inferred

Remember the `north/2` predicate in slide 36?

- it can be used in a "generative" way
  - e.g. `?- north(C, cell(2, 2, _)).` meaning "what's the cell `C` northward to `(2,2)`?"
  - e.g. `?- north(C1, C2).` meaning "let `C1` and `C2` be two cells such that `C1` is northward to `C2`"
- try these queries after removig the `cell(X,Y1,S1)` & `cell(X,Y2,S2)` subgoals from `north/2`. What changes?
  - try drawing the prooth-tree for both versions of `north/2`

The Need for Factorisation I

Remember the directions code from slide 36?

- some predicates share the same structure and simply differ for their *functors*

  e.g. `north/1`, `east/1`, `south_west/1`, ...
- writing them was repetitive and boring

We would need a way to factorise predicates having the same logic

- sadly, only constants can be used as functors

  i.e. we cannot write something like

  ```
  Op([C1, C2 | Cs]) :- Op(C1, C2), Op([C2 | Cs]).
  ```
- how could we "parametrize" the functor of some predicate?

The Need for Factorisation II

The power of building terms

- suppose we were able to dynamically create a term `T` (e.g. `T = a(X)`)
  - we could then add facts/rules to our theory by means of `assert(T)`
- suppose also we were able to call `assert(T)` on theory loading
  - we could then generate facts/rules programmatically, *without the need of typing them*

Building Terms Dynamically

The '=..'(?Term:any, ?List:list) operator definition

Term =.. List is true if List is a list consisting of the functor and all arguments of Term, in this order.

Examples for the '=..'/2 operator

```
?- a(b, c, d) =.. [a, b, c, d]. ⟶ yes.
?- a([X|Y], [A|B]) =.. [a, [X|Y], [A|B]]. ⟶ yes.
?- X =.. [a, b]. ⟶ X/a(b).
?- B =.. [b, c], X =.. [a, B]. ⟶ X/a(b(c)).
```

Directives

Prolog directives syntax

[tuProlog] engines support natively some directives, that can be defined by means of the :-/1 predicate in theory specification. Directives are used to specify properties of clauses and engines, format and syntax of terms.

The `initialization/1` directive

```
:- initialization(Goal).
```

- sets the goal to be executed just after the theory has been loaded

The `include/1` directive

```
:- include(FileName).
```

- loads the theory contained in the file specified by `FileName`

Meta-Programming I

Informal definition

Meta-programming is a programming technique in which computer programs have the ability to treat programs as their data, possibly by manipulating their own code while running

- this is made easy in Prolog, where both the program and its data share the same syntax

Meta-Programming II

Meta-programming example – The `binary_to_nary/1` predicate

```
binary_to_nary(Op) :-
  % op([X1, X2]) :- op(X1, X2).
  BaseCaseHead =.. [Op, [X, Y]],
  BaseCaseBody =.. [Op, X, Y],
  assertz(BaseCaseHead :- BaseCaseBody),
  % op([X1, X2 | Xs]) :- op(X1, X2), op([X2 | Xs]).
  RecursiveCaseHead =.. [Op, [X1, X2 | Xs]],
  RecursiveCaseBody1 =.. [Op, X1, X2],
  RecursiveCaseBody2 =.. [Op, [X2 | Xs]],
  assertz(RecursiveCaseHead :- (RecursiveCaseBody1,
    RecursiveCaseBody2)).

greater_than(X, Y) :- X > Y.

:- initialization(binary_to_nary(greater_than)).
```

Meta-Programming III

- `greater_than/2` is a binary predicate
  - the `greater_than/1` version, accepting an arbitrarily long list of numbers is attained by means of meta-programming

- try the following queries, with and without the `initialization/1` directive

```
?- greater_than(3, 2).
?- greater_than([4, 3, 2, 1]).
```

- try to make the Tic-Tac-Toe code more concise by means of the meta-programming technique.
  - a possible solution is available on the course site

Prolog Examples

Distributed Systems / Technologies

Sistemi Distribuiti / Tecnologie

Giovanni Ciatto
giovanni.ciatto@unibo.it

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018