11 Lab
Advanced Exercises in Prolog

Mirko Viroli, Gianluca Aguzzi
{mirko.viroli,gianluca.aguzzi}@unibo.it

C.D.L. Magistrale in Ingegneria e Scienze Informatiche
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2021/2022

Lab 11: Outline

- Other exercises in Prolog, mostly using cut
- Supporting a new data structure (graphs)
- An advanced exercise
  - Generation of TicTacToe tables

Part 1: basic cut operations

Ex 1.1: dropAny

```
% dropAny(?Elem,?List,?OutList)

dropAny(X, [X | T], T).
dropAny(X, [H | Xs], [H | L]) :- dropAny(X, Xs, L).
```

- Check the above code
- Drops any occurrence of element
  - dropAny(10,[10,20,10,30,10],L)
    - L/[20,10,30,10]
    - L/[10,20,30,10]
    - L/[10,20,10,30]

Part 1: basic cut operations

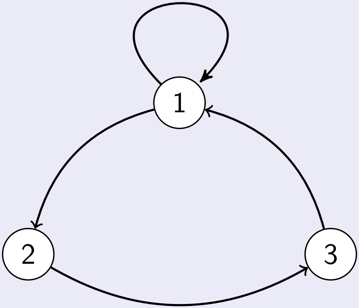
Ex1.1: other drops

- Implement the following variations, by using minimal interventions (cut and/or reworking the implementation)
  - `dropFirst`: drops only the first occurrence (showing no alternative results)
  - `dropLast`: drops only the last occurrence (showing no alternative results)
  - `dropAll`: drop all occurrences, returning a single list as result

Part 2: Operations on graphs

Model

- list of couples (e.g `[e(1,1),e(1,2),e(2,3),e(3,1)]`)
- the order of elements in the list is not relevant
- we use number to label nodes, but it could be anything



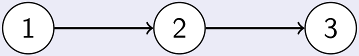
Part 2: Operations on graphs

Ex2.1: fromList

```
% fromList(+List,-Graph)

fromList([_],[]).
fromList([H1,H2|T],[e(H1,H2)|L]):- fromList([H2|T],L).
```

- Just analyse the code
- It obtains a graph from a list
  - `fromList([1,2,3],[e(1,2),e(2,3)]).`
  - `fromList([1,2],[e(1,2)]).`
  - `fromList([1],[]).`

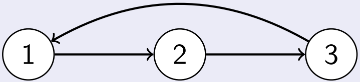


Ex2.2: `fromCircList`

```
% fromCircList(+List,-Graph)

% which implementation?
```

- Implement it!
  - `fromCircList([1,2,3],[e(1,2),e(2,3),e(3,1)]).`
  - `fromCircList([1,2],[e(1,2),e(2,1)]).`
  - `fromCircList([1],[e(1,1)]).`



Part 2: Operations on graphs

Ex2.3: inDegree

```
% inDegree(+Graph, +Node, -Deg)
%
% Deg is the number of edges leading into Node
```

- Implement it!
- `in_degree([e(1,2), e(1,3), e(3,2)], 2, 2).`
- `in_degree([e(1,2), e(1,3), e(3,2)], 3, 1).`
- `in_degree([e(1,2), e(1,3), e(3,2)], 1, 0).`

Ex2.4: dropNode

```
% dropNode(+Graph, +Node, -OutGraph)

% drop all edges starting and leaving from a Node
% use dropAll defined in 1.1??

dropNode(G,N,OG):- dropAll(G,e(N,_),G2),
      dropAll(G2,e(_,N),GO).
```

- Analyse this predicate
- `dropNode([e(1,2),e(1,3),e(2,3)],1,[e(2,3)]).`

Part 2: Operations on graphs

Ex2.5: reaching

```
% reaching(+Graph, +Node, -List)

% all the nodes that can be reached in 1 step from Node
% possibly use findall, looking for e(Node,_) combined
% with member(?Elem,?List)
```

- Implement it
- `reaching([e(1,2),e(1,3),e(2,3)],1,L). -> L/[2,3]`
- `reaching([e(1,2),e(1,2),e(2,3)],1,L). -> L/[2,2]).`

Ex2.6: anypath (advanced!!)

```
% anypath(+Graph, +Node1, +Node2, -ListPath)

% a path from Node1 to Node2
% if there are many path, they are showed 1-by-1
```

- `anypath([e(1,2),e(1,3),e(2,3)],1,3,L).`
  - `L/[e(1,2),e(2,3)]`
  - `L/[e(1,3)]`
- Implement it!
- Suggestions:
  - a path from N1 to N2 exists if there is a e(N1,N2)
  - a path from N1 to N2 is OK if N3 can be reached from N1, and then there is a path from N2 to N3, recursively

Part 2: Operations on graphs

Ex2.7: allreaching

```
% allreaching(+Graph, +Node, -List)

% all the nodes that can be reached from Node
% Suppose the graph is NOT circular!
% Use findall and anyPath!
```

- Implement it using the above suggestions
- `allreaching([e(1,2),e(2,3),e(3,5)],1,[2,3,5]).`

Ex2.8: grid-like nets

- During last lesson we see how to generate a grid-like network. Adapt that code to create a graph for the predicates implemented so far.
- Try to generate all paths from a node to another, limiting the maximum number of hops

Part 3: Generating Connect3 ("forza 3")

Ex3.1: next

- Implement predicate next/4 as follows
  - `next(@Table,@Player,-Result,-NewTable)`
  - Table is a representation of a TTT table where players x or o are playing
  - Player (either x or o) is the player to move
  - Result is either win(x), win(o), nothing, or even
  - NewTable is the table after a valid move
  - Should find a representation for the Table
  - Calling the predicate should give all results

Ex3.2: game

- Implement `game(@Table,@Player,-Result,-TableList)`
- TableList is the sequence of tables until Result win(x), win(o) or even

Part 3: Generating Connect3 ("forza 3")

Hints

- Choosing the right representation for a table is key
  - with a good representation it is easier to select the next move, and to check if somebody won
  - if needed, prepare to separate representation from visualisation
- Possibilities
  - `[[_,_,_],[x,o,x],[o,x,o]]`: nice but advanced
  - `[[n,n,n],[x,o,x],[o,x,o]]`: compact, but need work
  - `[cell(0,1,x),cell(1,1,o),cell(2,1,x),...]`: easier
  - ... do you have a different proposal?

Part 4: play with resolution (advanced!)

Source code to simulate resolution

```
% An example theory for simple resolution
rule(a, []).     % means:    a.
rule(b, []).     % means:    b.
rule(d, []).     % means:    d.
rule(c, [a,b]). % means:    c :- a,b.
rule(c, [a,d]). % means:    c :- a,d.
rule(c, [c]).   % means:    c :- c.
rule(d, [d]).   % means:    d :- d.
rule(d, []).     % means:    d.

% next(+RI, -RO) relates a resolvent with all the next ones
% a resolvent is a list of goals
% e.g.: next([c,a], R) -> R/[a,b,a]; R/[a,d,a]; R/[c,a]
next([G|T], R) :- rule(G,B), append(B,T,R).

% trace(+RI, -LR) relates a resolvent with all success trace
% a success trace is a list of resolvents, ending with []
% e.g.: trace([c], L)
% -> L/[[c],[a,b],[b],[]]; L/[[c],[a,d],[d],[]]; ...
trace([], [[]]).
```

Part 4: play with resolution (advanced!)

Ex4.1: `trace` is a sort of mini-Prolog interpreter

- actually, more expressive than you think...
- try to implement variations such that:
  - traces are given in opposite order than one would expect
  - if because of a loop a trace is becoming longer than 100, it is just discarded
  - solutions are explored breadth firts, not depth first