

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. 2024/2025

Lab 11: Outline

- Exercises with built-in numbers and lists
- Other exercises in Prolog, mostly using cut
- Supporting a new data structure (graphs)
- An advanced exercise
 - ▶ Generation of TicTacToe tables

Part 1: On built-in lists and numbers

- Implement new (or reimplement wrt previous lab) the following predicates, using built-in predicates for lists and numbers

Ex1.1: search2

```
1 % search2(Elem, List)
2 % looks for two consecutive occurrences of Elem
```

Ex1.2: search_two

```
1 % search_two(Elem, List)
2 % looks for two occurrences of Elem with any element in
   between!
```

- ▶ `search_two(a, [b,c,a,a,d,e]).` → no
- ▶ `search_two(a, [b,c,a,d,a,d,e]).` → yes

Part 1: On built-in lists and numbers

Ex1.3: size

```
1 % size(List, Size)
2 % Size will contain the number of elements in List
```

- is it fully relational?

Ex 1.4: sum

```
1 % sum(List, Sum)
2
3 ?- sum([1,2,3], X).
4 yes.
5 X/6
```

Part 1: On built-in lists and numbers

Ex1.5: max and min

```
1 % max(List,Max,Min)
2 % Max is the biggest element in List
3 % Min is the smallest element in List
4 % Suppose the list has at least one element
```

Ex1.6: sublist

```
1 % split(List1, Elements, SubList1, SubList2)
2 % Splits a list into two sublists based on a given set
  % of elements.
3 % example: split([10,20,30,40,50],2,L1,L2). -> L1
  % / [10,20] L2/[30,40,50]
```

Part 1: On built-in lists and numbers

Ex1.7: rotate

```
1 % rotate(List, RotatedList)
2 % Rotate a list, namely move the first element to the
  end of the list.
3 % example: rotate([10,20,30,40], L). -> L/[20,30,40,10]
```

Ex 1.8 count_occurrences

```
1 % count_occurrences(Element, List, Count)
2 % Count is the number of times Element appears in List.
3 % example: count_occurrences(2,[1,2,3,2,4,2],Count). ->
  Count/3
```

Part 1: On built-in lists and numbers

Ex 1.9: dice

```
1 % dice(X)
2 % Generates all possible outcomes of throwing a dice.
3 % example: dice(X): X/1; X/2; ... X/6
```

Ex 1.10: three_dice

```
1 % three_dice(L).
2 % Generates all possible outcomes of throwing three
  dices
3 % example: three_dice(L). -> L/[1,1,3]; L/[1,2,2];...;L
  /[3,1,1]
```

Part 1: On built-in lists and numbers

Ex 1.11: distinct

```
1 % distinct(List, DistinctList)
2 % DistinctList contains all distinct elements from List.
3 % example: distinct([1,2,3,2,4,1],L). -> L/[1,2,3,4]
```


Part 2: basic cut operations

Ex 2.1: dropAny

```
1 % dropAny(?Elem,?List,?OutList)
2
3 dropAny(X, [X | T], T).
4 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 2: basic cut operations

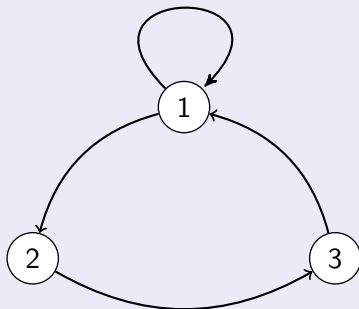
Ex2.2: other drops

- 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 a result

Part 3: 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 numbers to label nodes just as example



Part 3: Operations on graphs

Ex3.1: fromList

```
1 % fromList(+List,-Graph)
2
3 fromList([_],[]).
4 fromList([H1,H2|T],[e(H1,H2)|L]):- fromList([H2|T],L).
```

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



Part 3: Operations on graphs

Ex3.2: outDegree

```
1 % outDegree(+Graph, +Node, -Deg)
2 %
3 % Deg is the number of edges which start from Node
```

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

Part 3: Operations on graphs

Ex3.3: reaching

```
1 % reaching(+Graph, +Node, -List)
2
3 % all the nodes that can be reached in 1 step from Node
4 % possibly use findall, looking for e(Node,_) combined
5 % 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]) .`

Part 3: Operations on graphs

Ex3.4: nodes

```
1 % nodes (+Graph, -Nodes)
2 % craate a list of all nodes (no duplicates) in the
  graph (inverse of fromList)
```

- **Implement it**
- `nodes([e(1,2),e(2,3),e(3,4)],L) . -> L/[1,2,3,4]`
- `nodes([e(1,2),e(1,3)],L) . -> L/[1,2,3]`

Part 3: Operations on graphs

Ex3.5: anypath (advanced!!)

```
1 % anypath(+Graph, +Node1, +Node2, -ListPath)
2
3 % a path from Node1 to Node2
4 % 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 3: Operations on graphs

Ex3.6: allreaching

```
1 % allreaching(+Graph, +Node, -List)
2
3 % all the nodes that can be reached from Node
4 % Suppose the graph is NOT circular!
5 % Use findall and anyPath!
```

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

Ex3.7: grid-like nets (Optional)

- 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 4: Generating Connect3 (“forza 3”)

Ex4.1: 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

Ex4.2: game

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

Part 4: 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?