10 Lab
First 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 10: Outline

- The 2Prolog integration framework, many versions available
  - we adopt version 2p-4.0.3, available through `virtuale`
  - another version (newer, a little less stable) is
  - `https://github.com/tuProlog/2p-kt/releases/`
  - just double-click the jar and you are ready (should use JDK 11+)
  - or: java -jar *.jar from the console
- Be sure to let the teacher see each solution you produce, and to ask hints if something does not work or you get stuck!
- The following slides show what you should do
  - some examples are already implemented
  - others are for you to implement
- Red font means instructions for you!

Using the tuProlog GUI

- Type a Prolog theory/program in the *theory editor*
  - You can also type the theory in your favorite text editor and then cut and paste it on the *theory editor*
- Write a query in the *query text field* and press *Enter* (or push the *solve button*)
- The solution (if any) appears on the text area below
- Now you can take two different actions
  - Accept the obtained solution (push *Stop button*) or...
  - Search for other solutions (push *Solve button*)
- In case you want to generate all the possible solutions at once:
  - A fter typing a query, just push the *solve-all button*
  - The solutions appear on the same box as before
  - *Accept* and *Next* buttons are no longer active, as all the solutions have already been generated
- The text area on the bottom also features several tabs, not of interest today

Important Remark

- During this lab you will be asked several times to check whether a predicate is *fully relational* or not
- The meaning is:
  - Check whether the predicate works by using each argument both as input (with a ground term) and output (with a variable) – in case of predicates with N arguments, try with different combinations of the arguments
  - A term is said “ground” if it is fully instantiated, i.e., it includes no variable

Part 0a: Intro

Recap resolution, see slides 20-21

Type this theory:

- `a.`
- `a.`
- `a :- d.`
- `b.`
- `c :- a, b.`
- `c :- c.`

Verify that:

- Goal: `:- b.` gives: `yes`
- Goal: `:- a.` gives: `yes; yes; no`
- Goal: `:- c.` gives: `yes; yes; yes; yes; ...`

Anticipate (writing dows the resolution tree as in slides) and then verify the output of the following goals' resolutions:

- `a, b`
- `c, d`

Part 0b: Intro

Recap resolution and matching

Type this theory:

- `a(1).`
- `a(X) :- b(X), b(X).`
- `b(1).`
- `b(2).`
- `c(X) :- b(X).`
- `c(X) :- b(X), c(X).`

Verify that:

- Goal: `:- b(Y).` gives: `Y/1; Y/2`
- Goal: `:- a(1).` gives: `yes; yes`
- Goal: `:- a(Y).` gives: `Y/1; Y/1`
- Goal: `:- a(2).` gives: `no`

Anticipate (writing dows the resolution tree as in slides) and then verify the output of the following goals' resolutions:

- `c(1)`
- `c(Y)`
- `b(Y), b(Z)`

Part 1: Queries on list

Ex1.1: search

```
% search(Elem, List)

search(X, cons(X, _)).
search(X, cons(_, Xs)) :- search(X, Xs).
```

- `X|Xs` is another usual naming schema for `H|T`
- Write by-hand these clauses in the theory editor
- The above theory represents the search functionality
  - also called `element` or `member`
- Read the code as follows:
  - search is OK if the element `X` is the head of the list
  - search is OK if the element `X` occurs in the tail `Xs`

Part 1: Queries on list

Ex1.1: search

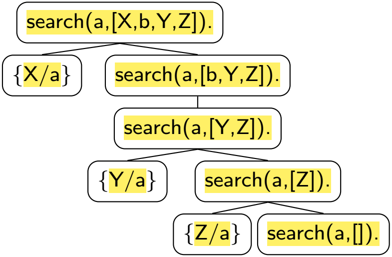
```
% search(Elem, List)

search(X, cons(X, _)).
search(X, cons(_, Xs)) :- search(X, Xs).
```

- `X|Xs` is another usual naming schema for `H|T`
- Write by-hand these clauses in the theory editor
- The above theory represents the search functionality
  - also called `element` or `member`
- Read the code as follows:
  - search is OK if the element `X` is the head of the list
  - search is OK if the element `X` occurs in the tail `Xs`

Part 1: Queries on list

- **One code, many purposes**
- Try the following goals:
  - Check all the possible solutions!
  - To this end, use either the solve-all button or the solve button: in the latter case, repeatedly use Next button until all the solutions are found
  - If you adopt solve-all be careful with infine branches in the resolution tree
- query:
  - `search(a, cons(a,cons(b,cons(c,nil))))`.
  - `search(a, cons(c,cons(d,cons(e,nil))))`.
- iteration:
  - `search(X, cons(a,cons(b,cons(c,nil))))`.
- generation:
  - `search(a, X)`.
  - `search(a, cons(X,cons(b,cons(Y,cons(Z,nil)))))`.
  - `search(X, Y)`.

Part 1: Resolution Tree of search



- The tree represents the computational behaviour: it is traversed in the so-called depth-first (left-most) strategy
  - which leads to the order of solutions `X/a`, `Y/a`, `Z/a`
- For pure convenience, we shall write in diagrams and prose e.g.:
  - `cons(a, cons(b, cons(c, nil)))` by `[a,b,c]`
  - don't do that in Prolog... for now

Part 1: Queries on list

Ex1.2: search2

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

search2(X, cons(X, cons(X, _))).
search2(X, cons(_, Xs)) :- search2(X, Xs).
```

- First predict and then test the result(s) of:
  - `search2(a, cons(c,cons(a,cons(a,cons(d,cons(a,cons(a,nil))))))).`
  - `search2(a, cons(c,cons(a,cons(a,cons(a,nil))))).`
  - `search2(a, cons(c,cons(a,cons(a,cons(b,nil))))).`
  - `search2(a, L).`
  - `search2(a, cons(_,cons(a,cons(_,cons(a,cons(_,nil)))))).`

Ex1.3: search_two

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

- Realise it yourself by changing search2, expected results are:
  - `search_two(a, cons(c,cons(a,cons(a,cons(b,nil))))).` → no
  - `search_two(a, cons(c,cons(a,cons(d,cons(a,cons(b,nil)))))).` → yes
- Check if it is fully relational

Part 1: Queries on list

Ex1.4: search_anytwo

```
% search_anytwo(Elem, List)
% looks for any Elem that occurs two times, anywhere
```

- Implement it
- Suggestion:
  - Elem must be on the head and search must be successful on the tail
  - otherwise proceed on the tail
  - (search_anytwo should use search)
- Expected results are:
  - `search_anytwo(a, cons(c,cons(a,cons(a,cons(b,nil))))).` → yes
  - `search_anytwo(a, cons(c,cons(a,cons(d,cons(a,cons(b,nil)))))).` → yes

Part 2: Extracting information from a list

Ex2.1: Peano size of a list

```
%%% size(List,Size)
% Size will contain the number of elements in List,
    written using notation zero, s(zero), s(s(zero))..
```

- Realise this version yourself!
  - `size(cons(a,cons(b,cons(c,nil))), X).` → `X/s(s(s(zero)))`
- Can it allow for a pure relational behaviour?
  - `size(L, s(s(s(zero)))).` ??
- **Note**: Built-in numbers are extra-relational!!

Part 2: Extracting information from a list

Ex 2.2: sum_list

```
% sum_list(List, Sum)

?- sum_list(cons(zero,cons(s(s(zero)),cons(s(zero),nil))), X).
yes.
X/s(s(s(zero)))
```

- Realise this version yourself, using sum of Peano numbers

Ex2.3: count (in tail-recursive fashion)

```
% count(List, Element, NOccurrences)
% it uses count(List, Element, NOccurrencesSoFar, NOccurrences)

count(List, E, N) :- count(List, E, zero, N).
count(nil, E, N, N).
count(cons(E, L), E, N, M) :- count(L, E, s(N), M).
count(cons(E, L), E2, N, M) :- E \= E2, count(L, E2, N, M).
```

- To realise this we need an “extra variable”
  - the usual “tail recursion schema”
  - we create new arguments and call a new predicate, which is count/4
- Check next slides, where we analise this solution

Part 2: Extracting information from a list

Ex2.3: count (resolution)

```
% count(List, Element, NOccurrences)
% it uses count(List, Element, NOccurrencesSoFar, NOccurrences)

count(List, E, N) :- count(List, E, zero, N).
count(nil, E, N, N).
count(cons(E, L), E, N, M) :- count(L, E, s(N), M).
count(cons(E, L), E2, N, M) :- E \= E2, count(L, E2, N, M).
```

- `count(cons(a,cons(b,cons(c,cons(a,cons(b,nil))))), a, N)`
- `count(cons(a,cons(b,cons(c,cons(a,cons(b,nil))))), a, zero, N)`
- `count(cons(b,cons(c,cons(a,cons(b,nil)))),a, s(zero), N)`
- `count(cons(c,cons(a,cons(b,nil))), a, s(zero), N)`
- `count([a,b], a, s(zero), N)`
- `count([b], a, s(s(zero)), N)`
- `count([], a, s(s(zero)), N)` → `N=s(s(zero))`

- Note: this is a tail recursion!!!

Part 2: Extracting information from a list

Ex2.3: count in Java

```java
int count(List l, int e){
  int count=0;
  for (;!l.isEmpty();l=l.getTail()){
    if (l.getHead() == e){
          count = count+1;
      }
  }
  return count;
}
```

- An iterative solution in Java using a class List with methods `isEmpty`, `getHead`, `getTail`

Part 2: Extracting information from a list

Ex2.3: count in Java (Recursive)

```java
int count(List l, int e) {
    return count(l, e, 0);
}
int count(List l, int e, int count) {
    if (l.isEmpty()) {
        return count;
    }
    if (l.getHead() == e) {
        count = count + 1;
    }
    return count(l.getTail(), e, count);
}
```

Part 2: Extracting information from a list

Ex2.3: count in Scala (Recursive)

```scala
def count(list: List[Int], e: int): Int =
  @tailrec
  def count(list: List[Double], e: Int, count: Int): Int =
    list match
      case ‘e‘ :: xs => count(xs, e, count + 1)
      case _ :: xs => count(xs, e, count)
      case _ => count
  count(list, e, 0)
```

Part 2: Extracting information from a list

Ex2.4: `max`

```
% max(List, Max)
% Max is the biggest element in List
% Suppose the list has at least one element
```

- Realise this yourself!
  - by properly changing count
- Do you need an extra argument?
  - first develop: `max(List,TempMax,Max)`
  - where `TempMax` is the maximum found so far (initially it is the first number in the list.)

Part 2: Extracting information from a list

Ex2.5: max and min

```
% min-max(List,Min,Max)
% Min is the smallest element in List
% Max is the biggest element in List
% Suppose the list has at least one element
```

- Realise this yourself!
  - by properly changing max
  - note you have a predicate with "2 outputs"

Part 3: Compare lists

Ex3.1: `same`

```
% same(List1,List2)
% are the two lists exactly the same?
```

- Predict and check relational behaviour!

Ex3.2: `all_bigger`

```
% all_bigger(List1,List2)
% all elements in List1 are bigger than those in List2,
    1 by 1
```

- example: `all_bigger(cons(s(s(zero)), cons(s(zero), nil)),cons(s(zero),cons(zero, nil))).`
- Do this yourself!

Part 3: Compare lists

Ex3.3: sublist

```
% sublist(List1,List2)
% List1 should contain elements all also in List2
```

- Do this yourself!
- Example:
  `sublist(cons(a,cons(b,nil)), cons(c,cons(b,cons(a,nil)))).`
    - do a recursion on List1, each time just use search of exercise 1.1!

Part 4: Creating lists

Ex4.1: seq

```
% seq(N,E,List) --> List is [E,E,...,E] with size N
% example: seq(s(s(s(zero))), a, cons(a,cons(a,cons(a,
    nil)))).

seq(zero, _, nil).
seq(s(N), E, cons(E,T)) :- seq(N, E, T)
```

- Check this implementation.
  - Is it fully relational?

Ex4.2: seqR

```
% seqR(N,List)
```

- Realise it yourself!
- example:
  `seqR(s(s(s(zero))), cons(s(s(zero)),cons(s(zero),cons(zero,nil)))).`

Ex4.3: seqR2

```
% seqR2(N,List) --> is [0,1,...,N-1]
```

- Realise it yourself!
- example: `seqR2(s(s(s(zero))), cons(zero,cons(s(zero),cons(s(s(zero)),nil)))).`
- Note, you may need to add a predicate "last"
  - `last(cons(a,cons(b,nil)),c,cons(a,cons(b,cons(c,nil)))).`

Part 5: Port list functions

- Consider few known list functions, how would you port them in Prolog? For each:
  - Write a small specification as Prolog comment
  - Implement it
  - Write as Prolog comment few usages
- Examples inspired by Scala:
  - (assume `l` is a List[Int])
  - `l.last, l map (_+1), l filter (_>0)`
  - `l count (_>0), l find (_>0)`
  - `l dropRight (2), l dropWhile (_>0)`
  - `l partition (_>0), l.reversed`
  - `l drop (2), l take (2), l.zip(l2)`