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. 2021/2022

Lab 10: Outline

- The 2Prolog integration framework, many versions available
  - we adopt version 0.20.4 of 2p-kt
  - http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome
  - 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  - *Accept* and *Next* buttons are no longer active, as all the solutions have already been generated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 1: Queries on list

Ex1.1: search

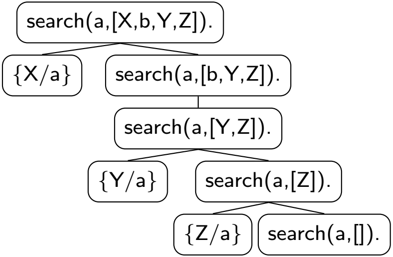
```
% search(Elem, List)

search(X,[X|_]).
search(X,[_|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/2`
  - called `member/2` in prolog library
- 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,[a,b,c]).`
  - `search(a,[c,d,e]).`
- iteration:
  - `search(X,[a,b,c]).`
- generation:
  - `search(a,X).`
  - `search(a,[X,b,Y,Z]).`
  - `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`

Part 1: Queries on list

Ex1.2: search2

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

search2(X,[X,X|_]).
search2(X,[_|Xs]) :- search2(X,Xs).
```

- First predict and then test the result(s) of:
  - search2(a,[b,c,a,a,d,e,a,a,g,h]).
  - search2(a,[b,c,a,a,a,d,e]).
  - search2(X,[b,c,a,a,d,d,e]).
  - search2(a,L).
  - search2(a,[_,_,a,_,a,_]).

Part 1: Queries on list

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

Part 2: Extracting information from a list

Ex2.1: size

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

size([],0).
size([_|Xs],N) :- size(Xs,N2), N is N2 + 1.
```

- Check whether it works!
- Can it allow for a fully relational behaviour?

Part 2: Extracting information from a list

Ex2.2: size with s(s(..(zero))

```
% 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([a,b,c],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 2.3: sum

```
% sum(List,Sum)

?- sum([1,2,3],X).
yes.
X/6
```

- Realise this version yourself!

Part 2: Extracting information from a list

Ex2.4: `average`

```
% average(List,Average)
% it uses average(List,Count,Sum,Average)

average(List,A) :- average(List,0,0,A).
average([],C,S,A) :- A is S/C.
average([X|Xs],C,S,A) :-
  C2 is C+1,
  S2 is S+X,
  average(Xs,C2,S2,A).
```

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

Part 2: Extracting information from a list

Ex2.4: average (resolution)

```prolog
% average(List,Average)
% it uses average(List,Count,Sum,Average)

average(List,A) :- average(List,0,0,A).
average([],C,S,A) :- A is S/C.
average([X|Xs],C,S,A) :-
  C2 is C+1,
  S2 is S+X,
  average(Xs,C2,S2,A).
```

- Sequence of resolvent/goals
  - `average([3,4,3],A)`
  - `average([3,4,3],0,0,A)`
  - `average([4,3],1,3,A)`
  - `average([3],2,7,A)`
  - `average([],3,10,A)` → A=3.3333
- Note: this is a tail recursion!!!

Part 2: Extracting information from a list

Ex2.4: average in Java

```java
int average(List l){
  int sum=0;
  int count=0;
  for (;!l.isEmpty();l=l.getTail()){
    count=count+1;
    sum=sum+l.getHead();
  }
  return sum/count;
}
```

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

Part 2: Extracting information from a list

Ex2.4: average in Java (Recursive)

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

Part 2: Extracting information from a list

Ex2.4: average in Scala (Recursive)

```scala
def average(list: List[Double]): Double =
  @tailrec
  def average(list: List[Double], count: Int, sum: Double): Double =
    list match
      case Nil      => sum / count
      case x :: xs => average(xs, count + 1, sum + x)

  average(list, 0, 0)
```

Part 2: Extracting information from a list

Ex2.5: maximum

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

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

Part 2: Extracting information from a list

Ex2.6: max and min

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

- Realise this yourself!
  - by properly changing max
  - note you ahve a predicate with “2 outputs”

Part 3: Compare lists

Ex3.1: `same`

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

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

- Predict and check relational behaviour!

Ex3.2: `all_bigger`

```prolog
% all_bigger(List1,List2)
% all elements in List1 are bigger than those in List2, 1 by 1
% example: all_bigger([10,20,30,40],[9,19,29,39]).
```

- Do this yourself!

Part 3: Compare lists

Ex3.3: `sublist`

```prolog
% sublist(List1,List2)
% List1 should contain elements all also in List2
% example: sublist([1,2],[5,3,2,1]).
```

- Do this yourself!
  - do a recursion on List1, each time just use search of exercise 1.1!

Part 4: Creating lists

Ex4.1: seq

```
% seq(N,List)
% example: seq(5,[0,0,0,0,0]).

seq(0,[]).
seq(N,[0|T]) :- N2 is N - 1, seq(N2,T).
```

- Check this implementation.
  - Is it fully relational?

Ex4.2: seqR

```
% seqR(N,List)
% example: seqR(4,[4,3,2,1,0]).
```

- Realise it yourself!

Part 4: Creating lists

Ex4.3: seqR2

```
% seqR2(N,List)
% example: seqR2(4,[0,1,2,3,4]).
```

- Realise it yourself!
- Note, you may need to add a predicate "last"
    - `last([1,2,3],5,[1,2,3,5]).`

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)`