

Practical Session E2

First Exercises in Prolog *Prolog (1)*

a.a. 2011/2012

Stefano Benedettini

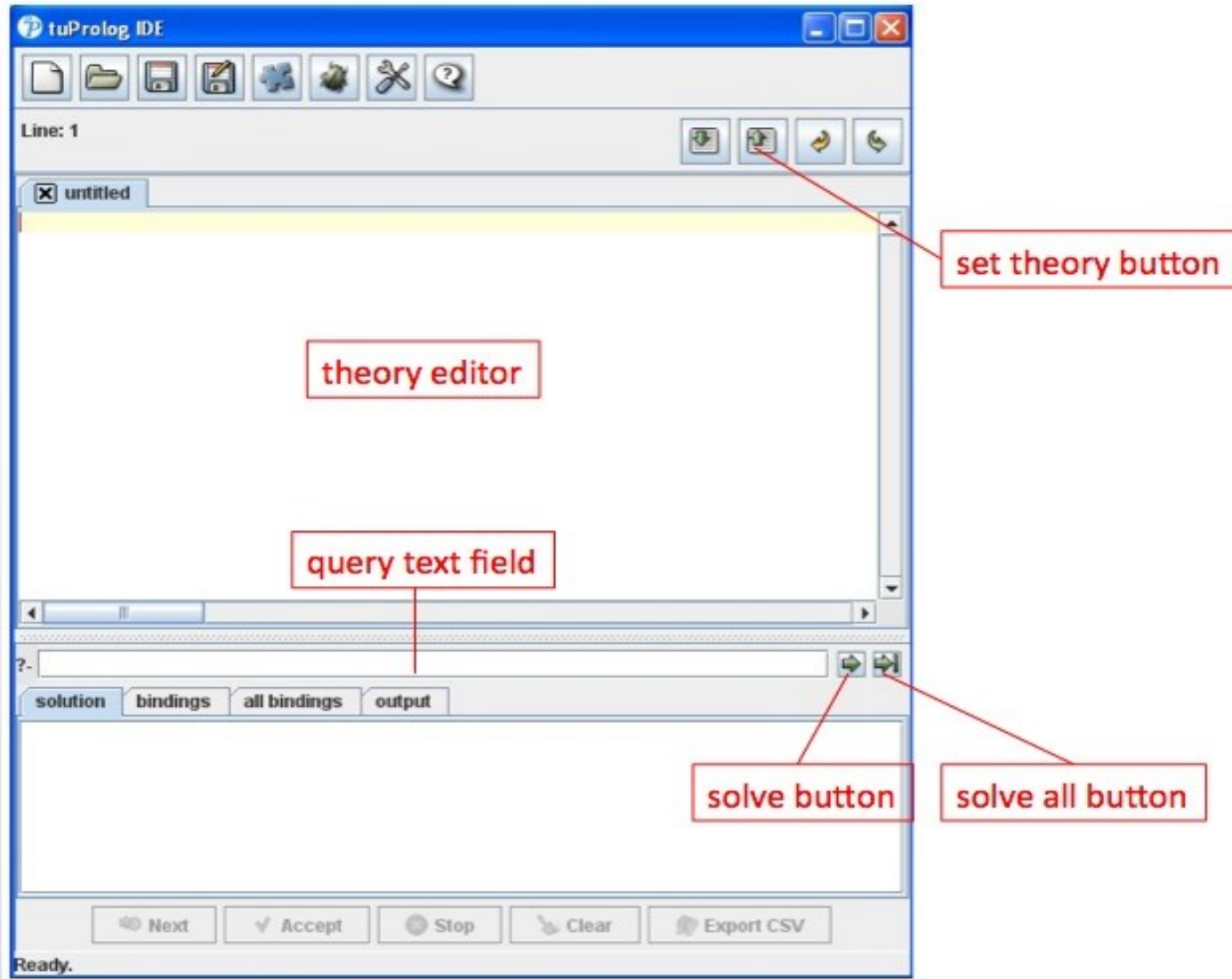
Outline

- The 2Prolog integration framework, many versions available
 - We adopt version 2.4 (with a caveat)
 - It is available on the Desktop
 - You can also get it from the website of the lab...
 - ...along with some documentation
- Just double-click on the jar and you are ready to go

Basic Instructions

- Login on a PC
- The following slides show what you should do
 - some examples are already implemented
 - others are for you to implement
- Red text means instructions for you!

The 2p GUI



Using the 2p GUI

- Type a Prolog theory 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 *Accept button*) or ...
 - Search for other solutions (push *Next button*)

More on Using the 2p GUI

- In case you want to generate all the possible solutions at once:
 - After 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 - *Accept* and *Next* buttons are no longer active, as all the solutions have already been generated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 to see the bindings generated over a resolution, ...
- The toolbar on the top also provides buttons for saving theories on the file systems, as well as for loading theories from file

Important Remark

- During this lab you will be asked several times to check whether a predicate is *fully relational* or not:
 - You have not been introduced yet to this concept
- For now, the meaning is:
 - Check whether the predicate keeps behaving as it should even though the meaning of the arguments is changed, i.e. by using each argument both as input and output for the predicate (in case of predicates with N arguments, try with different combinations of the arguments)

Part 1

- Queries on list
- Given a list, check whether it satisfies some properties!

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*
 - You can also type the theory in a text editor and save it as a *.pl file, then load the file in the 2p GUI by exploiting the *load theory* functionality
- The above theory represents the search functionality
- 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

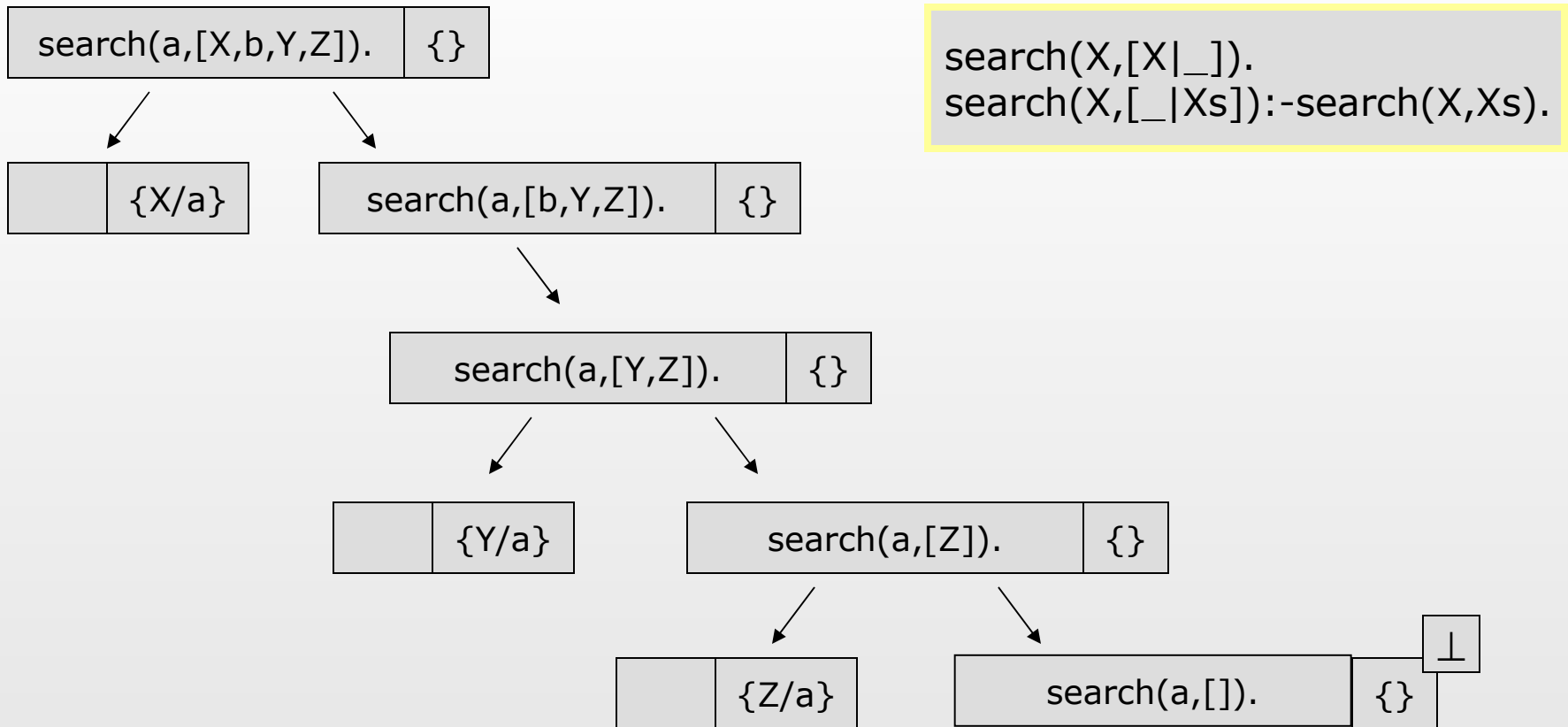
1 code, many purposes

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 infinite 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).`

Note: 1 code for different purposes!

Resolution Tree



- The tree is traversed in the so-called depth-first (left-most) strategy
 - which leads to the order of solutions X/a, Y/a, Z/a

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,_,_]).

Ex1.3: search_two

```
% search_two(Elem,List)
% looks for two occurrences of Elem..
% .. with an 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
 - search_two(a,[b,c,a,a,a,a,e]). → yes (2 solutions)
- Check if it is fully relational

Ex1.4: search_anytwo

```
% search_anytwo(Elem,List)
% looks for any Elem that occurs (at least) twice
```

- **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
- Given a list, reporting some information on it!

Ex2.1: size

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

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

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

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!
 - $\text{size}([a,b,c],X) \rightarrow X/s(s(s(zero)))$
- Can it allow for a pure relational behaviour?
 - $\text{size}(L, s(s(s(zero))))$. ??
- Note: Built-in numbers are extra-relational!!

Ex2.2a: conversion NAT $\rightarrow$ int

- Write two predicates that convert NATs to built-in integers and vice-versa:
 - `toint(s(s(s(zero))),3).` $\rightarrow$ yes
 - `tonat(4,s(s(s(s(zero))))).` $\rightarrow$ yes

Ex 2.3: sum and prod

```
% sum(List,Sum)
% prod(List,Product)

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

?- prod([1,2,3,4],X).
yes.
X/24
```

- Realise them yourself!
- How prod is different than sum?

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”
 - we create new arguments and call a new helper predicate, which is average/4
- Check next slides, where we analyse this solution

Ex2.4: average (resolution)

```
average(List,A) :- average(List,0,0.0,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 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!!!

Ex2.4: average in Java

```
int average(List l){  
    double 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

Ex2.4: average in Java (ric)

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

```
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).
```

- Recursive variant

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 3

- Compare lists
- Given two lists, do some computation!

Ex3.1: all_bigger

```
% 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!

Ex3.2: sublist

```
% sublist(List1,List2)  
% List1 should be a subset of List2  
% example: sublist([1,2],[5,3,2,1]).
```

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

Ex3.3: reversed

```
% reversed(List1,List2)
% Is List2 List1 reversed?

?- reversed([a,b,c,d,e],[e,d,c,b,a]).
yes.
```

- Predict and check relational behaviour
- Provide a tail-recursive version (no append allowed)

Part 4

- Creating lists

Ex4.1: seq

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

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

- Check this implementation.
 - Is it fully relational?
 - Use NATs instead of integers

Ex4.2: seqR

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

- **Realise it yourself!**
 - Try to aim for a tail-recursive version
 - You can't use reverse/2 built-in predicate in any case

Ex4.3: seqF

```
% seqF(N,List)
% example: seqF(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]).
- Again, try also a tail-recursive version

Other Exercises

```
% replicate_n(X,Int,List)
% ?- replicate_n(a,3,[a,a,a]). → yes
% ?- replicate_n(X,0,[ ]). → yes
```

```
% double(List,List)
% suggestion: remember predicate append/3
% ?- double([1,2,3],[1,2,3,1,2,3]). → yes
```

```
% times(List,N,List)
% ?- times([1,2,3],3,[1,2,3,1,2,3,1,2,3]). → yes
```

```
% proj(List,List)
% ?- proj([[1,2],[3,4],[5,6]],[1,3,5]). → yes
```