Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #3

CS@TUCN

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Agenda

- Example 4 – length of a list
  - introducing recursion types
  - Types of parameters
- Example 5 – reversing a list
  - introducing pattern position
  - Meaning of arguments
- Backtracking prevention – the CUT (!)
  - Negation (as failure)

Example 4 – how to recurse?

- Determine the length of a list
- Number of arguments?

```
list_length/2 (list, length)
list_length([],0).                  //length of empty list is 0
list_length([_|T],N):-
      list_length(T,N1),            //length of tail calculated as N1
      N is N1+1.                    //add 1 to the length of tail
```

- Order of clauses? This way? Swapped? Which is correct? Which is better? Why? Is it important? Why?

(discuss indexation on the first argument. Oral explanation.)

- Meaning of partial result (N1) not quite relevant (is length of the tail)
- What if we need to keep the number of "consumed" elements?

Example 4 – how to recurse?

- Determine the length of a list
- Number of arguments?

```
list_length/2 (list, length)
list_length([],0).                  //length of empty list is 0
list_length([_|T],N):-
      list_length(T,N1),            //length of tail calculated as N1
      N is N1+1.                    //add 1 to the length of tail
```

- Order of clauses? This way? Swapped? Which is correct? Which is better? Why? Is it important? Why?

(discuss indexation on the first argument. Oral explanation.)

- Meaning of partial result (N1) not quite relevant (is length of the tail)
- What if we need to keep the number of "consumed" elements?

List length – backward recursion

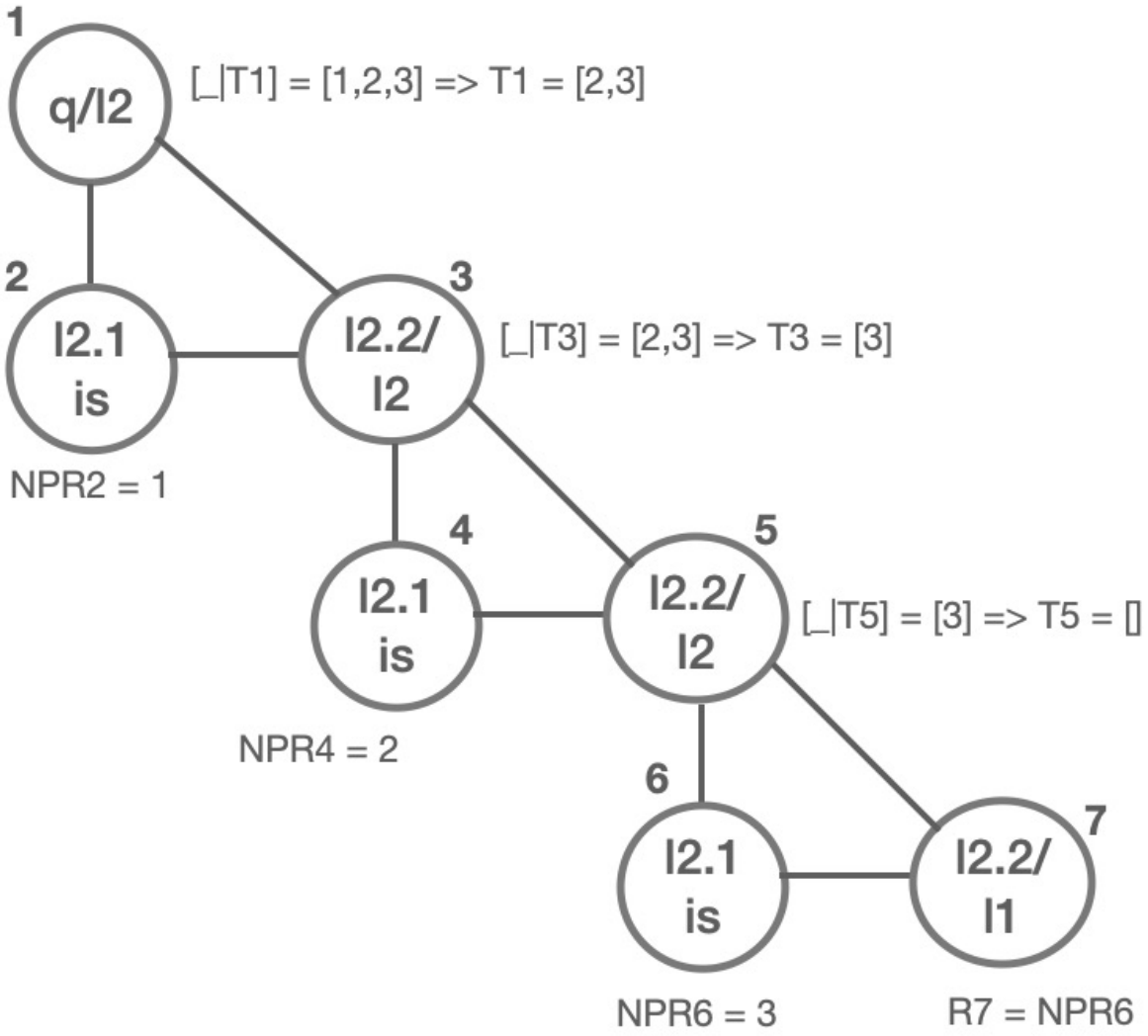
- Previous solution uses ***backward*** recursion, meaning that the **result is built as recursion RETURNS**
- What if we build it while advancing?

```
list_length1/2 (length, list)
list_length1(PR,[_|T]):-
        NPR is PR+1,        //add 1 to the partial result so far
                            //i.e. length of the covered part
        list_length1(NPR, T).//go count the rest of the elements
list_length1(R,[]).         // R has the final list length
```

- what's the difference compared to the previous solution?
- N represents here what we *have covered* ; before – what we've *yet to cover*.

list_length1 deduction tree

```
l1 list_length1([],R).
l2 list_length1([_|T],PR):-
        NPR is PR+1,          l2.1
        list_length1(T,NPR).  l2.2
q ?- list_length1([1,2,3], 0).
```



List length – forward recursion

`list_length1/2 (list, length)`

- Not enough – as result, although calculated, is NEVER reported

```
list_length1([],N).
list_length1([_|T],N):-
        N is N1+1,
        list_length1(T,N1).
```

- Upgrade it with 3rd argument = "steal" the result from the rightmost leaf

```
list_length3([],PR,FR):-FR=PR.
list_length3([_|T],PR,FR):-
        NPR is PR+1,
        list_length3(T,NPR,FR).
```

List length – forward recursion

- With default unification:

```
list_length3([],Rez,Rez).
list_length3([_|T],PR,FR):-
      NPR is PR+1,
      list_length3(T,NPR,FR).
```

- How should we call it? Needs specific INITIALIZATION:

```
?- list_length3(InList,0,Rezult).
```

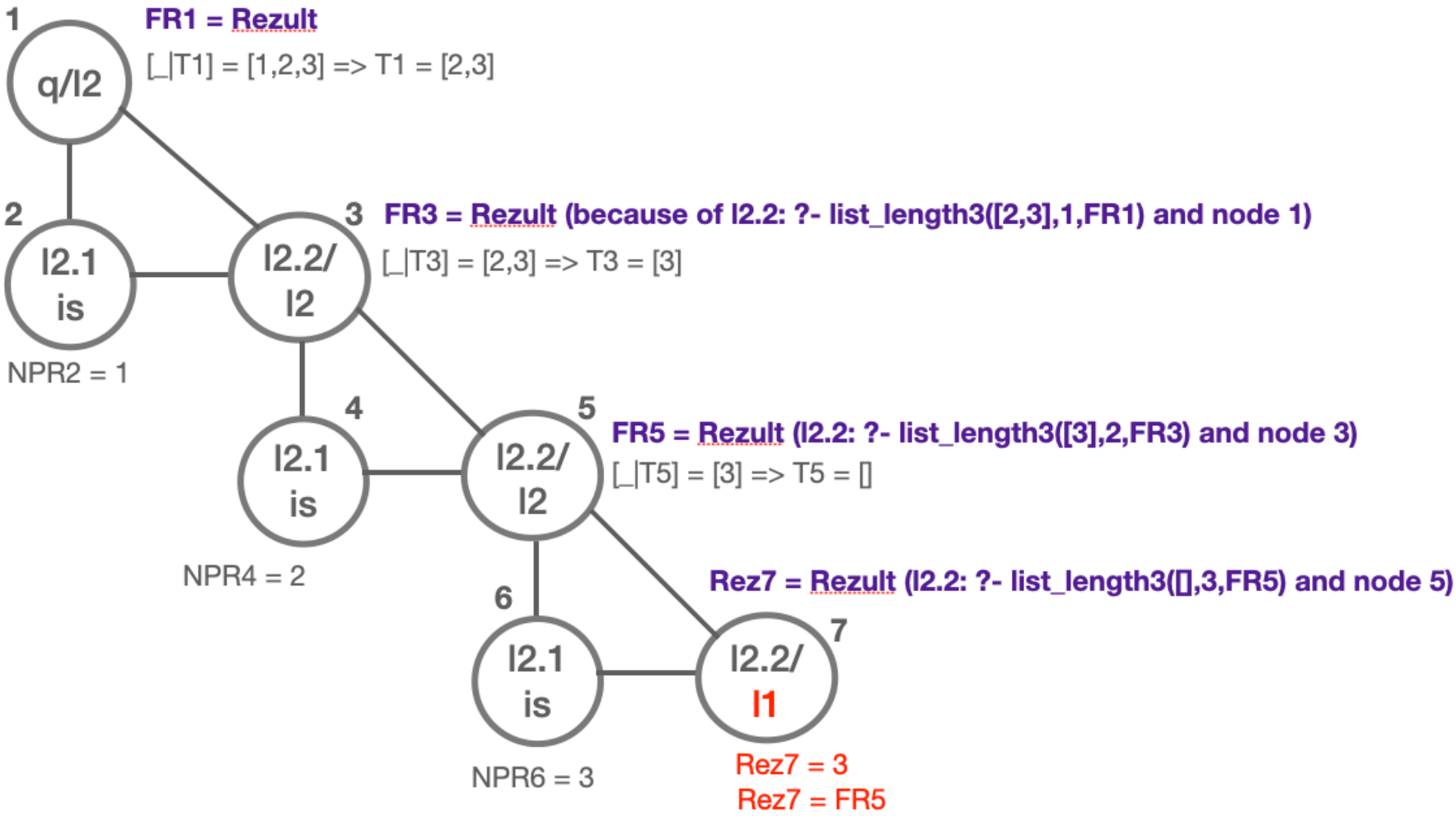
- To avoid mandatory initial call, use a wrapper:

```
list_length3(InList,Rezult):-
      list_length3(InList,0,Rezult).
```

- Can have same name? why/why not?
- Discuss signature definition

list_length3 deduction tree

```
11 list_length3([],Rez,Rez).
12 list_length3([_|T],PR,FR):-
     NPR is PR+1,                    12.1
     list_length3(T,NPR,FR).         12.2
q ?- list_length3(InList,0,Rezult).
```



TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Forward vs Backward recursion

	Forward recursion	Backward recursion
+	- Potentially useful partial results - Efficiency if parallelized - LCO (Last Call Optimization)	- No need for extra variable - No need for special call
-	- Need for extra variable (final result) - Need for specialized call (for accumulator initialization)	- Rarely useful intermediate results - Not efficient if parallelized (execution of concurrent process(es) - delayed until recursion returns)

Reversing a list

- How many arguments?
  - 2 = list + reversed list

```
reverse1/2 (list, reversed_list)
reverse1([],[]).      //empty input returns empty output
reverse1([H|T],R):-
      reverse1(T,PR),  //reverses the tail of list
      append(PR,[H],R). //concatenates the partial results
```

- Needs specialized call? Why/why not?
- What type of recursion is here?
  - *Backward*
- Efficiency? How is it calculated?
- Can we avoid 2 traversals through the list? How?

Reversing a list – forward recursion

- How many arguments?
  list + partial_rev_list+reversed list

```
reverse2/3(list,part_rev_list,rev_list)
reverse2([],PR,PR).  //when input gets empty the partial
                     //result becomes the final one
reverse2([H|T],PR,R):-
     NPR=[H|PR],          //concatenates the partial results
                          //PR, NPR acts as a stack
     reverse2(T,NPR,R).   //reverses the tail of list
```

- Wrapper (MUST have)

```
reverse(In,Out):-
     reverse2(In,[],Out).
```

- Efficiency? Is it better?

Reversing a list – forward recursion

```
reverse2([],PR,PR).
reverse2([H|T],PR,R):-
      reverse2(T,[H|PR],R).
```

- Same as before just default unification what was explicit
- Let's change the order of arguments 2 and 3 (makes no difference in the functionality)

```
reverse2([],PR,PR).
reverse2([H|T],R,PR):-
      reverse2(T,R,[H|PR]).
```

- Just variables renaming:

```
reverse2([],L,L).
reverse2([H|T],L,R):-
      reverse2(T,L,[H|R]).
```

Comparative analysis

```
reverse([],L,L).
reverse([H|T],L,R):-
     reverse(T,L,[H|R]).
```

```
append([],L,L).
append([H|T],L,[H|R]):-
     append(T,L,R).
```

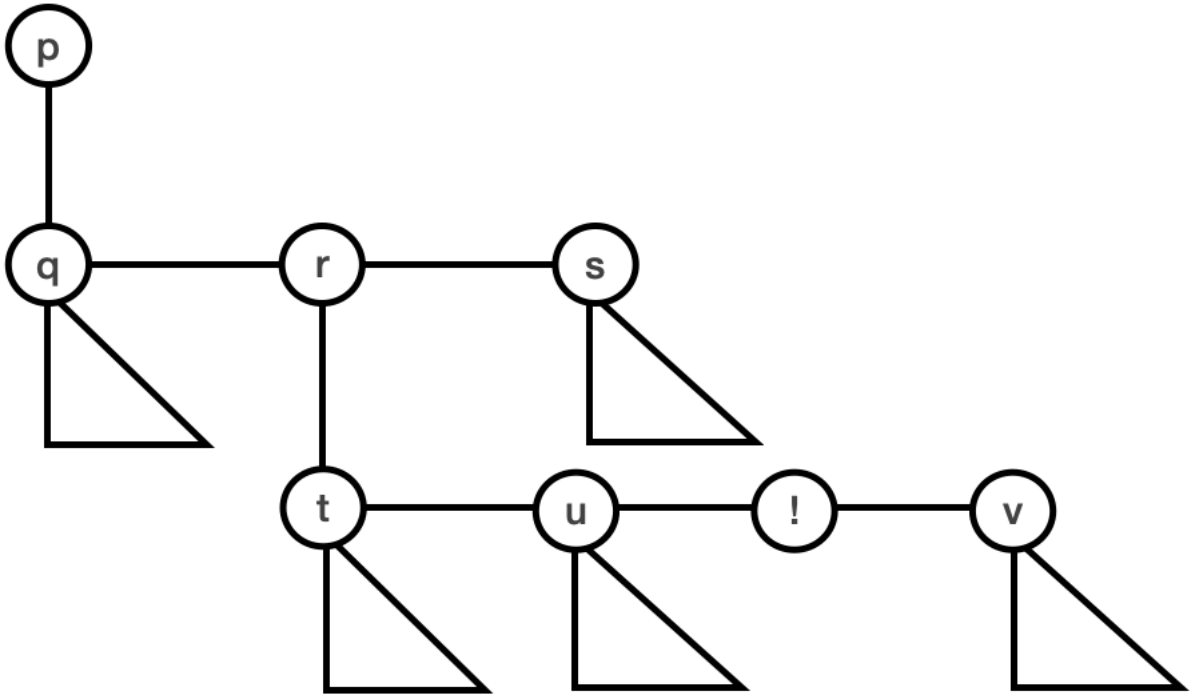
- What is the difference?
- Meaning of pattern's position
- Length of arguments analysis

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Backtracking avoidance (The CUT!)

- Built-in predicate
- Always succeeds
- Never backtracks

```
p:-q,r,s.
r:-t,u,!,v.
```



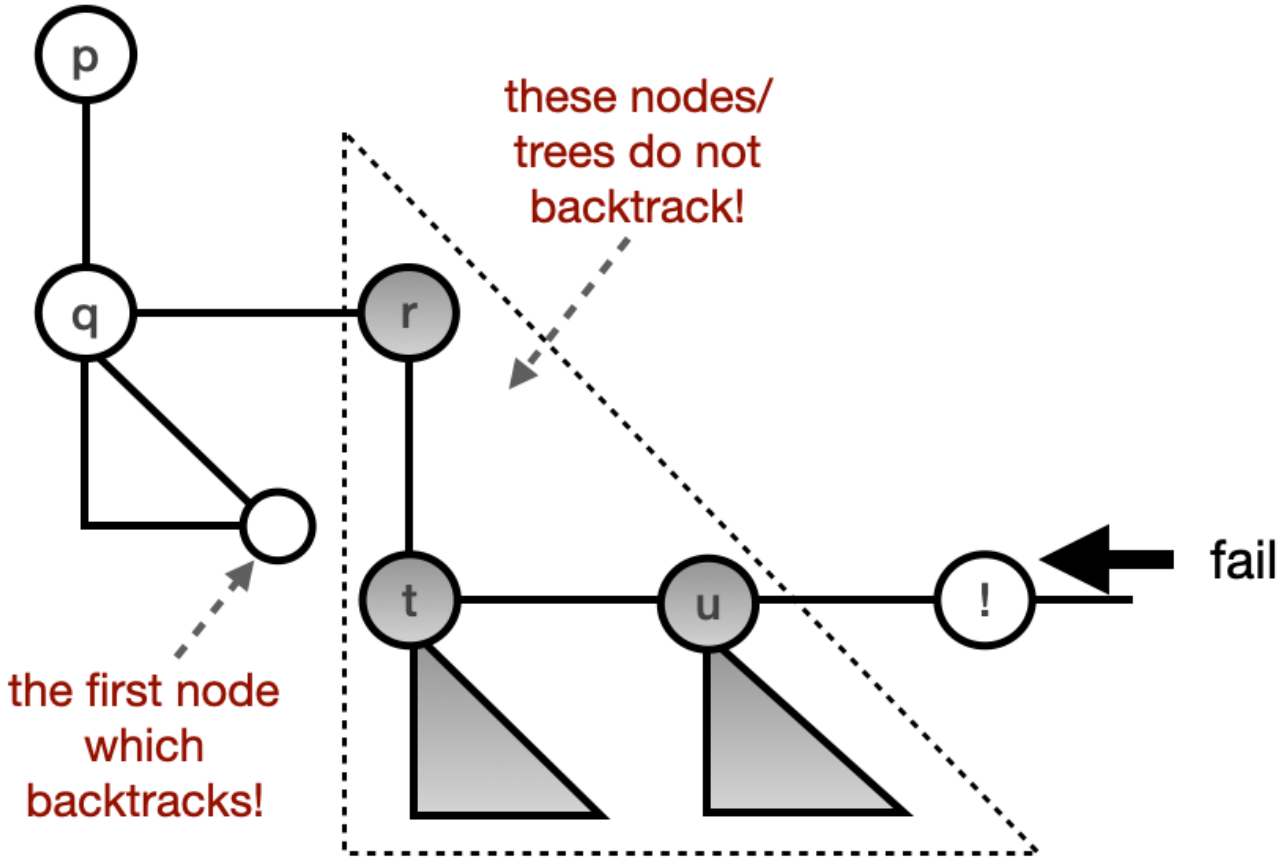
TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

The CUT (!) – effects

- ! Prevents from backtracking the following categories:
  - All left siblings
  - All subtrees of left sibling nodes
  - Parent node
- NO other node is affected:
  - Siblings on the right/their subtrees have the ability to backtrack
  - Left siblings DO backtrack BEFORE ! Is executed
  - Their subtrees also
- As everything, takes effect ONLY AFTER its (!) execution (before execution it does NOT actually exist)

The CUT (!)

```
p:-q,r,s.
r:-t,u,!,v.
```



Negation (as failure)

- Unfair behavior
- As it just assumes the **inability to prove T means F**
- Isn't it so? Why? Postponed discussion.

```
not(P):-
      P,!, fail.
not(P).
```

- P – true => not(P) – false; OK.
- P – false => not(P) – true; OK
- P – ? Don't know how it is. Why? Incomplete information, not known at the time.

  How is not(P)? Let's run it:

  P's execution fails (P is NOT yet T),
  we enter the second clause
  succeeds.

Negation (as failure)

```
not(P):-
      P,!, fail.
not(P).
```

- Negation assumes negated is T just because we don't have information about P.
- Negation as failure (to check validity)