Lecture 10: Recursion in Prolog

Recursive data structures and relations. Applications

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

April 26, 2021

A notion is recursive if it is defined in terms of itself.

- We can define recursive data types, recursive functions, or recursive relations.
- In general, a recursive definition consists of
  - 0 or more base cases
  - 1 or more recursive cases

In Prolog

- we can work with recursive data: trees, lists, etc.
- we can define recursive relations

List = predefined datatype of Prolog. A list is either

- The empty list `[]` (base case)
- `[term | list]` (recursive case)

Some definitions of predicates and relations for lists

```
% isList(lst) holds if lst is a list
isList([]).                         % base case
isList([_|T]):-isList(T).           % recursive case

% member(term,lst) holds if term is in list lst
member(X,[X|_]).        % base case: term is the head of the list
member(X,[_|T]):-member(X,T).       % recursive case
```

Remarks

1. All variables must start with _ or with an uppercase letter.
2. Like in Haskell, we can use the anonymous variable _ which matches every term.
3. Variables can occur many times, both in the head and in the body of a clause. We could have written

   ```
   member(X,[Y|_]):-X=Y.
   ```

   but Prolog allows to be more concise, and write

   ```
   member(X,[X|_]).
   ```

4. `member` is predefined in Prolog.

Remarks

More about list membership tests

Prolog tries to find if a term is in a list by applying the rules in the order from the program:

1. `member(X,[X|T]).` (base case)
2. `member(X,[_|T]):-member(X,T).` (recursive case)

In the recursive case, the list gets shorter and shorter.

- The list can not be shortened indefinitely $\Rightarrow$ computation will terminate.

Prolog stops computation in 2 situations:

1. when it encounters a list for which base case holds $\Rightarrow$ it returns `true`.
2. when it reaches the empty list $\Rightarrow$ no more rules are applicable $\Rightarrow$ it returns `false`.

Quiz

What happens when Prolog is asked to answer the following queries:

```
?- member(X,[a,b,c]).
?- member(a,X).
?- member(X,Y).
?- member(X,_).
?- member(_,Y).
?- member(_,_).
```

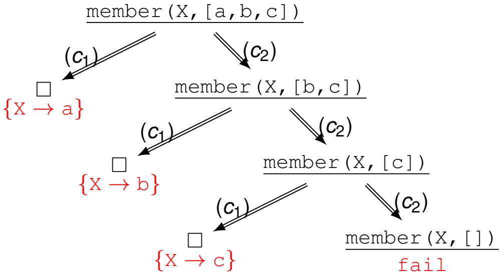
Note that some queries have multiple answers.

Quiz

What happens when Prolog is asked to answer the following queries:

```
?- member(X,[a,b,c]).
?- member(a,X).
?- member(X,Y).
?- member(X,_).
?- member(_,Y).
?- member(_,_).
```

Note that some queries have multiple answers.



We can work with binary trees defined by the grammar

```
btree ::= null | bt(string,btree,btree)
```

Some definitions of predicates and relations for binary trees

```
% isBT(bt) holds if bt is a binary tree
isBT(null).
isBT(bt(S,T1,T2)):-string(S),isBT(T1),isBT(T2).

% toList(bt,lst) holds if lst is the list of strings from bt
toList(null,[]).
toList(bt(S,T1,T2),Lst):-
    toList(T1,L1),
    toList(T1,L2),
    append(L1,[S|L2],Lst).
```

Termination = property of a program to stop after a finite number of computational steps.

- Some programs do not terminate

  ```
  parent(X,Y):-son(Y,X).
  son(Y,X):-parent(X,Y).
  ```

  Reason: these definitions are circular.
  $\Rightarrow$ avoid circular definitions!

- The following program does not terminate because it is is left-recursive:

  ```
  man(X):-man(Y),parent(X,Y).
  man(adam).
  ```

$\Rightarrow$ left-recursion should be used with care!

Program clauses (=rules and facts) are applied in the order in which they are written in the program.

- Intuitive criterion: facts should appear before rules.

Sometimes, rules ordered in a particular way work well only for a particular kind of queries.

Example

```
isList([_|T]):-isList(T).
isList([]).
```

is adequate to answer the queries

```
?-isList([1,2,3]).
?-isList([]).
?-isList(f(1,2)).
```

but inadequate for `?-isList(X).`

Program clauses (=rules and facts) are applied in the order in which they are written in the program.

- Intuitive criterion: facts should appear before rules.

Sometimes, rules ordered in a particular way work well only for a particular kind of queries.

Example

```
isList([_|T]):-isList(T).
isList([]).
```

is adequate to answer the queries

```
?-isList([1,2,3]).
?-isList([]).
?-isList(f(1,2)).
```

but inadequate for `?-isList(X).`

Question: What answer do you get to `?-isList(X)` if you change the order of clauses in the program?

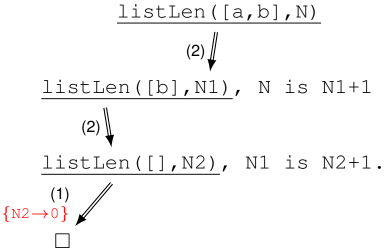
Many predefined predicates of Prolog make distinction between

Input parameters (-): they should have concrete values when the predicate is called.

Output parameters (+): their values are computed as answers to the query.

Arbitrary parameters (?): can be both input and output.

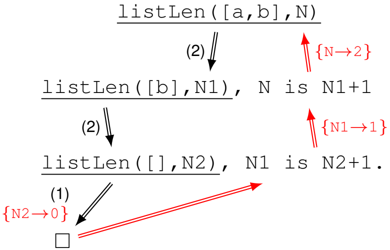
```
% listLen(+List,-N) computes the number N of elements of List.
% List is an input parameter, and N is output parameter.
listLen([],0).                                  % (1)
listLen([_|T],N):-listLen(T,N1), N is N1+1.     % (2)
```



Recursion

Example

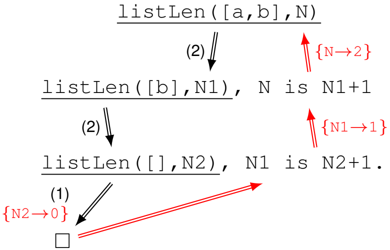
```
% listLen(+List,-N) computes the number N of elements of List.
% List is an input parameter, and N is output parameter.
listLen([],0).                                    % (1)
listLen([_|T],N):-listLen(T,N1), N is N1+1.       % (2)
```



Remarks:

Recursion
Example

```
% listLen(+List,-N) computes the number N of elements of List.
% List is an input parameter, and N is output parameter.
listLen([],0).                                      % (1)
listLen([_|T],N):-listLen(T,N1), N is N1+1.         % (2)
```

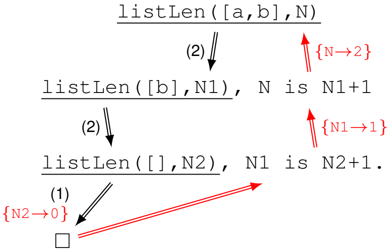


Remarks:

- The answer to the number N of elements is built on the branch of return from recursion

Recursion
Example

```
% listLen(+List,-N) computes the number N of elements of List.
% List is an input parameter, and N is output parameter.
listLen([],0).                                    % (1)
listLen([_|T],N):-listLen(T,N1), N is N1+1.       % (2)
```



Remarks:

- The answer to the number `N` of elements is built on the branch of return from recursion
- We wish to build the value of `N` on the branch which advances through recursion, to avoid the creation of temporary variables on the call stack.

`listLen1(List,N)` computes the number of elements in `List` on the branch which advances through recursion by using the auxiliary predicate `listLenAux(List,A,N)` where:

- `A` is a new argument, called accumulator.
- `A` accumulates the number of elements in the list while it advances through recursion.

```
listLen1(List,N):-listLenAux(List,0,N).                %1
listLenAux([],N,N).                                     %2
listLenAux([_|T],M,N):-P is M+1,listLenAux(T,P,N).     %3
```

```
elemLst1([a,b],N).
```

Recursion

Counting the number of elements in a list with an accumulator

`listLen1(List,N)` computes the number of elements in `List` on the branch which advances through recursion by using the auxiliary predicate `listLenAux(List,A,N)` where:

- `A` is a new argument, called accumulator.
- `A` accumulates the number of elements in the list while it advances through recursion.

```
listLen1(List,N):-listLenAux(List,0,N).                    %1
listLenAux([],N,N).                                        %2
listLenAux([_|T],M,N):-P is M+1,listLenAux(T,P,N).         %3
```

```
  elemLst1([a,b],N).
        ⇓ (1)
elemLstAux([a,b],0,N).
```

`listLen1(List,N)` computes the number of elements in `List` on the branch which advances through recursion by using the auxiliary predicate `listLenAux(List,A,N)` where:

- `A` is a new argument, called accumulator.
- `A` accumulates the number of elements in the list while it advances through recursion.

```
listLen1(List,N):-listLenAux(List,0,N).                    %1
listLenAux([],N,N).                                        %2
listLenAux([_|T],M,N):-P is M+1,listLenAux(T,P,N).         %3
```

```
   elemLst1([a,b],N).
          ⇓ (1)
elemLstAux([a,b],0,N).
          ⇓ (3)
  elemLstAux([b],1,N).
```

Recursion

Counting the number of elements in a list with an accumulator

`listLen1(List,N)` computes the number of elements in `List` on the branch which advances through recursion by using the auxiliary predicate `listLenAux(List,A,N)` where:

- `A` is a new argument, called accumulator.
- `A` accumulates the number of elements in the list while it advances through recursion.

```
listLen1(List,N):-listLenAux(List,0,N).                    %1
listLenAux([],N,N).                                         %2
listLenAux([_|T],M,N):-P is M+1,listLenAux(T,P,N).         %3
```

```
elemLst1([a,b],N).
      ⇓ (1)
elemLstAux([a,b],0,N).
      ⇓ (3)
elemLstAux([b],1,N).
      ⇓ (3)
elemLstAux([],2,N).
```

`listLen1(List,N)` computes the number of elements in `List` on the branch which advances through recursion by using the auxiliary predicate `listLenAux(List,A,N)` where:

- `A` is a new argument, called accumulator.
- `A` accumulates the number of elements in the list while it advances through recursion.

```
listLen1(List,N):-listLenAux(List,0,N).                  %1
listLenAux([],N,N).                                      %2
listLenAux([_|T],M,N):-P is M+1,listLenAux(T,P,N).       %3
```

```
elemLst1([a,b],N).
    ⇓ (1)
elemLstAux([a,b],0,N).
    ⇓ (3)
elemLstAux([b],1,N).
    ⇓ (3)
elemLstAux([],2,N).
    ⇓ (2)
□{N → 2}
```

Define the relation `revList(L,R)` to hold if `R` is the reverse of list `L`.

Main idea: Use an accumulator which acts like a stack where we push recursively all elements of `L`, starting with its head.
Initially, the accumulator is empty.

```
revList(L,R):-revListAux(L,R,[]).                 % (1)
% base case
revListAux([],R,R).                               % (2)
% recursive case
revListAux([H|T],R,A):-                           % (3)
   revListAux(T,R,[H|A]).
```

List reversal with accumulators

Illustrated example

```
?- revList([a,b,c],R).
```

List reversal with accumulators

Illustrated example

```
?- revList([a,b,c],R).
```

```
revList([a,b,c],R).
```

List reversal with accumulators

Illustrated example

```
?- revList([a,b,c],R).
```

$$\begin{array}{c}\texttt{revList([a,b,c],R).}\\ \Big\Downarrow \texttt{(1)}\\ \texttt{revListAux([a,b,c],R,[]).}\end{array}$$

List reversal with accumulators

Illustrated example

```
?- revList([a,b,c],R).
```

```
revList([a,b,c],R).
        ⇓ (1)
revListAux([a,b,c],R,[]).
        ⇓ (2)
revListAux([b,c],R,[a]).
```

List reversal with accumulators

Illustrated example

```
?- revList([a,b,c],R).
```

```
revList([a,b,c],R).
    ⇓ (1)
revListAux([a,b,c],R,[]).
    ⇓ (2)
revListAux([b,c],R,[a]).
    ⇓ (2)
revListAux([c],R,[b,a]).
```

List reversal with accumulators

Illustrated example

```
?- revList([a,b,c],R).
```

$$
\begin{array}{c}
\texttt{revList([a,b,c],R).} \\
\Big\Downarrow \text{(1)} \\
\texttt{revListAux([a,b,c],R,[]).} \\
\Big\Downarrow \text{(2)} \\
\texttt{revListAux([b,c],R,[a]).} \\
\Big\Downarrow \text{(2)} \\
\texttt{revListAux([c],R,[b,a]).} \\
\Big\Downarrow \text{(2)} \\
\texttt{revListAux([],R,[c,b,a]).}
\end{array}
$$

List reversal with accumulators

Illustrated example

```
?- revList([a,b,c],R).
R = [c,b,a].
```

```
revList([a,b,c],R).
        ⇓ (1)
revListAux([a,b,c],R,[]).
        ⇓ (2)
revListAux([b,c],R,[a]).
        ⇓ (2)
revListAux([c],R,[b,a]).
        ⇓ (2)
revListAux([],R,[c,b,a]).
        ⇓ (3)
        □
  {R → [c,b,a]}
```

Formalize the following knowledge in Prolog:

1. Stephen is neighbor of Peter.
2. Stephen is married with a doctor who works at emergency hospital.
3. Peter is married with an actress who works at the national theatre.
4. Stephen is melomaniac and Peter is hunter.
5. All melomaniacs are sentimental.
6. All hunters are liars.
7. Actresses like sentimental people.
8. Married people have same neighbors.
9. The relations of being married and being neighbors are symmetric.

Then, use Prolog to find the answer to the following question: Does Peter's wife like Stephen?

Neighbors problem

Knowledge representation in Prolog

```
neighbor1(stephen,peter).                                    %1
married1(stephen,wife_stephen).                              %2
doctor(wife_stephen).                                        %2
works(wife_stephen,emergencyHospital).                       %2
married1(peter,wife_peter).                                  %3
actress(wife_peter).                                         %3
works(wife_peter,nationalTheatre).                           %3
melomaniac(stephen).                                         %4
hunter(peter).                                               %4
sentimental(X):-melomaniac(X).                               %5
liar(X):-hunter(X).                                          %6
likes(X,Y):-actress(X),sentimental(Y).                       %7
neighbor(X,Y):-married(X,Z),neighbor(Z,Y).                   %8
neighbor(X,Y):-neighbor1(X,Y).                               %9
neighbor(X,Y):-neighbor1(Y,X).                               %9
married(X,Y):-married1(X,Y).                                 %9
married(X,Y):-married1(Y,X).                                 %9
conclusion:-married(peter,Wife),likes(Wife,stephen).

?-conclusion.
```

Remark: This program is recursive.

Remarks about symmetric relations

A binary relation `r` is symmetric if

`r(term`$_1$`,term`$_2$`)` holds if and only if `r(term`$_2$`,term`$_1$`)` holds.

The relations `neighbor` and `married` from the previous example are symmetric.

Q: How can we specify a symmetric relation?

Remarks about symmetric relations

A binary relation `r` is symmetric if

> `r(term`$_1$`,term`$_2$`)` holds if and only if `r(term`$_2$`,term`$_1$`)` holds.

The relations `neighbor` and `married` from the previous example are symmetric.

Q: How can we specify a symmetric relation?

- Version 1 - Example

  ```
  r(a,b).  r(a,c).
  r(X,Y):-r(Y,X).
  ```

  Remark: It is essential to place the facts for `r` before the rule for `r`.
  Problem:

  ```
  ?-r(b,c).
  ```

  $\Rightarrow$ this query will never be answered (infinite computation).
  How can we avoid these infinite computations?

Remarks about symmetric relations

A binary relation `r` is symmetric if

`r(term`$_1$`,term`$_2$`)` holds if and only if `r(term`$_2$`,term`$_1$`)` holds.

The relations `neighbor` and `married` from the previous example are symmetric.

Q: How can we specify a symmetric relation?

- Version 1 - Example

  ```
  r(a,b).  r(a,c).
  r(X,Y):-r(Y,X).
  ```

  Remark: It is essential to place the facts for `r` before the rule for `r`.
  Problem:

  ```
  ?-r(b,c).
  ```

  $\Rightarrow$ this query will never be answered (infinite computation).
  How can we avoid these infinite computations?

- Version 2: by using an asymmetric binary relation `r1`. For example:

  ```
  r1(a,b).  r1(a,c).
  r(X,Y):-r1(X,Y).
  r(X,Y):-r1(Y,X).
  ```

Remarks about symmetric relations

A binary relation `r` is symmetric if

`r(term`$_1$`,term`$_2$`)` holds if and only if `r(term`$_2$`,term`$_1$`)` holds.

The relations `neighbor` and `married` from the previous example are symmetric.

Q: How can we specify a symmetric relation?

- Version 1 - Example

  ```
  r(a,b).  r(a,c).
  r(X,Y):-r(Y,X).
  ```

  Remark: It is essential to place the facts for `r` before the rule for `r`.
  Problem:

  ```
  ?-r(b,c).
  ```

  $\Rightarrow$ this query will never be answered (infinite computation).
  How can we avoid these infinite computations?

- Version 2: by using an asymmetric binary relation `r1`. For example:

  ```
  r1(a,b).  r1(a,c).
  r(X,Y):-r1(X,Y).
  r(X,Y):-r1(Y,X).
  ```

  This version was used to define the symmetric relations `neighbor` and `married`.

Representation of sets in Prolog

A set can be represented as a list in which every element occurs only once.

- Define recursively the property `isSet(L)` to hold if `L` is a list in which every element occurs only once. For example:

```
?-isSet([a,b,d,c]).
true .
?-isSet([a,b,a]).
false .
```

- Define the relation `toSet(L,M)` which takes as input argument a list `L` and uses `M` as output parameter for the rest of elements that occur in `L`.

```
?-toSet([a,b,a,c],M).
M=[a,b,c]
```

Representation of sets in Prolog

1. `isSet(L)`
   - ▷ Base case: `[]` is set.
   - ▷ Recursive case: `[H|T]` is set if `H` does not occur in `T` and `T` is set.
2. `toSet(L,M)`
   - ▷ Base case: If `L=[]` then `M=[]`.
   - ▷ Recursive case: If `L=[H|T]` then `M=[H|R]` where `R` is the list produced in 2 steps:
     1. First, find list `R1` produced from `T` by removing all occurrences of `H`.
        To compute `R1`, we can define relation `del(H,T,R1)` to hold if `R1` is the list produced from `T` by removing all occurrences of `H`.
     2. `R` is produced recursively, as answer to the query `toSet(R1,R).`

Representation of sets in Prolog

```
isSet([]).
isSet([H|T]):-not(member(H,T)), isSet(T).

toSet([],[]).
toSet([H|T],[H|R]):-del(H,T,R1),toSet(R1,R).

del(H,[],[]).
del(H,[H|T],R):-del(H,T,R).
del(H,[H1|T],[H1|R]):-H\=H1,
                      del(H,T,R).
```

In Peano arithmetic, natural numbers are represented by terms defined by

```
nat ::= 0 | s(nat)
```

For example, `s(s(0))` represents number `2`. Define the following relations on numbers represented in Peano arithmetic:

```
% add(+X,+Y,-Z) holds if Z is the sum of X and Y
% mul(+X,+Y,-Z) holds if Z is the product of X and Y
% gt(+X,+Y) holds if X is strictly greater than Y.
```

Consider the relation `next(X,Y,L)` defined by:

```
next(X,Y,[X,Y|_]).
next(X,Y,[Z|T]):-next(X,Y,T).
```

1. What is the meaning of `next(X,Y,Z)`?
2. What is the meaning of `z_u(X,Y)` defined by the rule

```
z_u(X,Y):-next(X,Y,[monday,tuesday,wednesday,
                    thursday,friday,
                    saturday,sunday,monday]).
```

3. What is the meaning of `z_u(X,Y)` defined by the rule

```
z_p(X,Y):-z_u(Y,X).
```