2

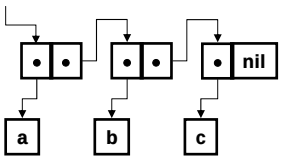
Programming with Logic and Constraints

Roman Barták
Charles University, Prague (CZ)

`roman.bartak@mff.cuni.cz`
`http://ktiml.mff.cuni.cz/~bartak`

Lists

- How to represent a **list of elements**?
- Using terms:
  - a pointer-like structure
  - list(a,list(b,list(c,nil)))



- Prolog provides this structure directly:
  - `[Head|Tail]`
  - `[a,b,c] =[a|[b|[c|[]]]]`
  - Elements can be anything, e.g. a list again
    - `[[q,2], 12, f(a,b), [[]]]`
- This is a syntactic sugar only!

Membership

- How to check **membership in a list**?
- Explore the list from start until the element is found.

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

?-member(a,[a,b,a]). -> yes
?-member(X,[a,b,a]). -> X=a; X=b; X=a
?-member(a,L).       -> L=[a|_]; L=[_,a|_], …
```

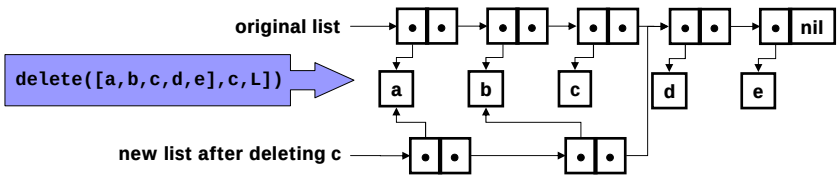

Deleting element

- Delete the first occurrence of X from the list.
  `delete(List,X,ListWithoutX)`

```
delete([],_,[]).
delete([X|T],X,T).
delete([Y|T],X,[Y|NewT]):-
    X\=Y, delete(T,X,NewT).
```

X and Y cannot be unified

- The part of the list before X is duplicated!



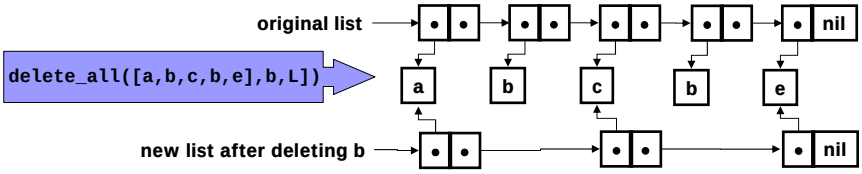
Deleting

- Delete all occurrences of X from the list.

```
delete_all([],_X,[]).
delete_all([X|T],X,NewT):-
  delete_all(T,X,NewT).
delete_all([Y|T],X,[Y|NewT]):-
  X\=Y,delete_all(T,X,NewT).
```

difference from delete

- The list is completely duplicated in memory.



Inserting

- Insert X before the list `insert(L,X,LStartWithX)`:

```
insert(L,X,[X|L]).
```

- Add X to the end of the list `add(L,X,LEndWithX)` :

```
add([],X,[X]).
add([Y|T],X,[Y|NewT]):-
  add(T,X,NewT).
```

  - Again, the list is completely duplicated!
  - The procedure can also remove the last element from the list!

```
?-add(NewList,X,[a,b,c,d]).
NewList=[a,b,c]
X=d
```

Concatenating

- concatenate two lists
  - **concat(L1,L2,L)**
  - L1=[a,b,c], L2=[d,e] -> L=[a,b,c,d,e]

```
concat([],L,L).
concat([H|T],L2,[H|NewT]):-
  concat(T,L2,NewT).
```

- Time and space complexity depends on the size of the first list!
- The procedure can also be used to split the list.

```
?-concat(List1,List2,[a,b,c,d]).
List1=[], List2=[a,b,c,d] ;
List1=[a], List2=[b,c,d] ;
…
```


Reverting

- Revert the list
  - **revert(L,Rev)**
  - L=[a,b,c] -> Rev=[c,b,a]

```
revert([],[]).
revert([H|T],Rev):-
  revert(T,RT),
  add(RT,H,Rev).
```

Slow and memory consuming!

Try to omit **add** (**concat**) in your code.

Much better solution is using **accumulator**!

```
revert1(List,Rev):-
  rev(List,[],Rev).
rev([],L,L).
rev([H|T],Acc,Rev):-
  rev(T,[H|Acc],Rev).
```

list length	revert	revert1
50000	39 s.	0 s.

Operators

- writing everything as a term is not always comfortable
  - compare '='(X,'+'(2,3)) and X=2+3
- a more human readable form of terms would be appropriate
  - e.g. **infix notation of "standard" operations** (provided by Prolog)
- moreover, user may define **own operators** via
  `:- op(precedence, type, name).`
- this is only a **"syntactic sugar"**

Arithmetic expressions

```
?-X=1+2.   -> X=1+2
?-3=1+2.   -> no
```

Number is a special type of atom. It has a semantics (it is a number)!

- Term 1+2 is different from the term 3.
  - No semantics is associated with terms!
- We need a special procedure to evaluate the numerical expression: "is"

```
?-X is 1+2.
X=3
```

- `X is Expr` works as **arithmetic evaluator**:
  - evaluate Expr and compare (unify) the result with X
- Be careful: **"is" is not an assignment** command!

```
?-X is 1+2, X is 7.
```

Arithmetic comparison

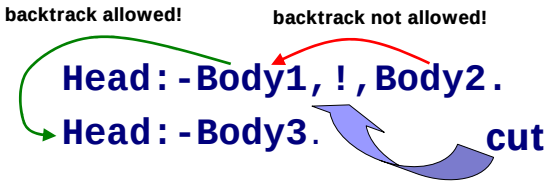
- If we have numbers, can we compare them?
- Prolog provides standard comparison of numbers:
  - `X < Y`
    - The numeric value of X is less than the numeric value of Y

```
?-1<2.      -> yes
?-1+1<3.    -> yes
?-3<1+2.    -> no
```

  - `X > Y, X =< Y, X >= Y`

Cut

- Prolog uses **depth-first search** to cover **non-determinism** of alternative rules.
  - use choice point when there is an alternative
- Can we **prune alternatives explicitly**?
  - **Cut** removes the choice point so no alternative rules will be tried.



Practicing cuts

```
test1(X,Y):-
  member(Y,[[1,2],[3,4]]),member(X,Y).
test1(0,[]).

test2(X,Y):-
  !,member(Y,[[1,2],[3,4]]),member(X,Y).
test2(0,[]).

test3(X,Y):-
  member(Y,[[1,2],[3,4]]),!,member(X,Y).
test3(0,[]).

test4(X,Y):-
  member(Y,[[1,2],[3,4]]),member(X,Y),!.
test4(0,[]).
```

Examples of red cuts

Their usage is discouraged because they change computation!

X	1	2	3	4	0
Y	[1,2]	[1,2]	[3,4]	[3,4]	[]

Cut for determinism

- Prune branches that will not be visited (**green** cut).

Example:

split the list into a list with elements smaller than X and a list with elements not smaller than X

```
split([],_,[],[]):-!.
split([H|T],X,[H|T1],T2):-
  H<X,!,
  split(T,X,T1,T2).
split([H|T],X,T1,[H|T2]):-
  split(T,X,T1,T2).
```

Negation

- How to prove non-existence of the solution?
- Useful for complex tests like non-member.

```
\+ :Goal
```

- no variable binding!
- **Inside negation:**

```
not(Query):-
  call(Query),!,fail.
not(_Query):-
  true.
```

META-PREDICATE
Prolog goal is a term so any term can be used as a query

If Query succeeds then fail (cut forbids using the alternative rule), otherwise succeed using the alternative rule.

Practicing negation

- Negation in Prolog is **negation-as-failure**
  - It is not a full logical negation!

```
p(a).
p(b).
q(a).
?- \+ (p(X),q(X)), X=b.    -> fail
?- X=b, \+ (p(X),q(X)).    -> X=b
```

- **Be especially careful when negation is applied to non-ground goal (contains variables)!**

All solutions

- How to find all answers to a Query?

```
findall(?Template,:Query,?List)
```

Collects all answers to Query in the form of Template in a List.

Example:

Find all neighboring nodes of “a”.

```
?-findall(X,edge(a,X),Neigborhood).
?-findall(f(X),edge(a,X),Neigborhood).
?-findall(dzzz,edge(a,X),Neigborhood).
```

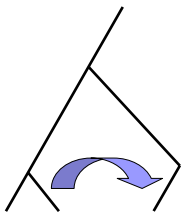
[b,c]

[f(b),f(c)]

[dzzz,dzzz]

Blackboard

- How to pass information back when backtracking?
- How to pass information between search branches?
- We can use the Prolog database!
  - **`assert`** the information in one branch
  - access it in the other branch
- It is better to use **blackboard**!
  - clear and efficient



Blackboard primitives

- Each information stored in the blackboard is identified by a unique atom called a **key** (an atom defined by the user).

- **`bb_put(:Key, +Term)`**
- **`bb_get(:Key, ?Term)`**
- **`bb_delete(:Key, ?Term)`**
- **`bb_update(:Key, ?OldTerm, ?NewTerm)`**

Blackboard example

- Test satisfiability of Query without binding variables.

```
sat(Query, _Answer):-
  bb_put(sat,no),
  once(Query), % finds one solution (if any)
  bb_put(sat,yes),
  fail.
sat(_Query,Answer):-
  bb_delete(sat,Answer).
```

Another solution using negation and if-then-else:

```
sat2(Query,Answer):-
  (\+ call(Query) -> Answer=no ; Answer=yes).
```

Practicing blackboard

- Count the number of answers to Query **`sat_num(:Query,-NumAnswers)`**

```
sat_num(Query,_NumAnswers):-
  bb_put(counter,0),
  call(Query),
  bb_get(counter,N),
  N1 is N+1,
  bb_put(counter,N1),
  fail.
sat_num(_Query,NumAnswers):-
  bb_delete(counter,NumAnswers).
```

```
arc(a,b).
arc(a,c).
arc(a,d).

?-sat_num(arc(a,X),N).
N=3;
no
```

- Another solution using **`findall`**:

```
sat_num(Query,NumAnswers):-
  findall(x,Query,List),
  length(List,NumAnswers).
```


Blackboard features

- Blackboard works as a global „variable“.
- **Be careful of nesting!**
  - If Query in the previous examples calls **`sat`** then the blackboard data are mishandled.
- Structure of the term is preserved but a connection to the „local“ variables is lost!!

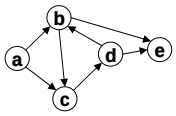
```
?-A=term(X,f(X)), bb_put(test,A), X=a,
  bb_get(test,B).
A = term(a,f(a)),
B = term(_A,f(_A)),
X = a ? ;
no
```

Final PROLOG practice

Compute (one of) the shortest path between two nodes (avoid cycling).

- **Database (graph):**

```
arc(a,b).
arc(a,c).
arc(b,c).
arc(b,e).
arc(c,d).
arc(d,b).
arc(d,e).
```



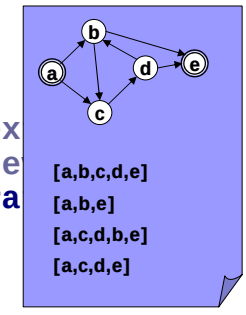
- **Expected answers:**

```
?-shortest_path(a,a,P).
  P = [a]

?- shortest_path(a,e,P).
  P = [a,b,e]

?- shortest_path(e,b,P).
  no
```


Shortest path (naïve)

- Find **all paths in a DFS manner** and then select the shortest.

```
shortest_path(From,To, ShortestPath):-
  findall(Path,path(From,To,[],Path),AllPaths),
  shortest_list(AllPaths,ShortestPath).

path(From,From,Visited,Path):-!,
  revert([From|Visited],Path).
path(From,To,Visited,Path):-
  arc(From,Through),          % nex
  \+ member(Through,Visited), % pre
  path(Through,To,[From|Visited],Pa
```



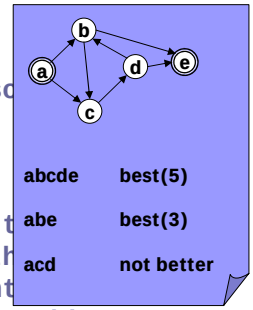
Shortest path (B&B)

- **Branch&Bound** exploring all paths in a DFS manner

```
shortest_pathBB(From,To,_Path):-
  bb_put(best,no_path),
  spathBB(From,To,[],0).
shortest_pathBB(_From,_To,Path):-
  bb_get(best,path(_,Path)).
```

```
can_be_shorter(_):-
    bb_get(best,no_path).
can_be_shorter(Length):-
    bb_get(best,path(BestLength,_)),
    Length<BestLength.
```

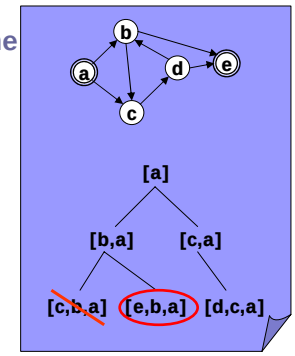
```
spathBB(From,From,Visited,Length):-!,
  revert([From|Visited],Path),
  bb_put(best,path(Length,Path)),% save so
  fail.
spathBB(From,To,Visited,OldLength):-
  NewLength is OldLength+1,
  can_be_shorter(NewLength),      % check t
  arc(From,Through),              % find th
  \+ member(Through,Visited),     % prevent
  spathBB(Through,To,[From|Visited],NewLength).
```



Shortest path (BFS)

- **Breadth-first search** with concatenation

```
shortest_pathBFS(From,To,Path):-
  spathBFS([[From]],To,Path).

spathBFS([Visited|Rest],To,Path):-
  Visited = [N|_],
  (N=To ->                    % we found the path
      revert(Visited,Path)
  ;                           % expand the
      findall([N1|Visited],
          (arc(N,N1),
            \+ member(N1,Visited),
            \+ member([N1|_],Rest)),
          NewNodes),
      concat(Rest,NewNodes, Nodes),
      spathBFS(Nodes,To,Path)
  ).
```



Homework

- Write procedures (rules) defining:
  - `length(List,Length)`
  - `shortest_list(ListOfLists,ShortestList)`
- Write a Prolog program solving the **water pouring problem**.
  - We have three (N) **cups**, each with a given **capacity** and a given **level** of water. It is possible to pour completely a cup into another cup (if capacity is not exceeded) or pour part of a cup to fill another cup. Find a **shortest plan** for reaching a given level of water in each cup.
  - Tip: use the shortest path algorithms!