

Universidade do Minho
Escola de Engenharia
Departamento de Informática

# Prolog Lists

Mestrado Integrado em Engenharia Informática Licenciatura em Engenharia Informática Inteligência Artificial





- Introduce lists, an important recursive data structure often used in Prolog programming;
- member/2 predicate, a fundamental Prolog tool for manipulating lists;
- Recursing lists.





- A list is a finite sequence of elements;
- Elements are enclosed in square brackets;
- Number of elements → length;
- List can have all sort of prolog elements;
- Empty list: [].





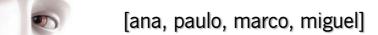
- **Example:**
- o [ana, paulo miguel, sara]
  - [ana, peluche(coelhinho), X, 2, ana, []]
  - o [ana, [miguel, juliana], [rosa, amigo(rosa)]]
  - [ ], feliz(z), [2, [b,c]], [ ], Z, [2, [b,c]]]





- A non-empty list consists of 2 parts:
  - o The head;
  - o The tail.
- Head → first item in the list;
- o Tail → everything else.
  - o tail is the list that remains when we remove the first element;
  - tail of a list is always a list!





Head → ana

Tail→ [paulo, marco, miguel]

[ [ ] , feliz(z), [2, [b,c]], [ ], Z, [b,c]]

 $Head \rightarrow []$ 

Tail  $\rightarrow$  [feliz(z), [2, [b,c]], [], Z, [b,c]]





[feliz(z)]

Head: feliz(Z)

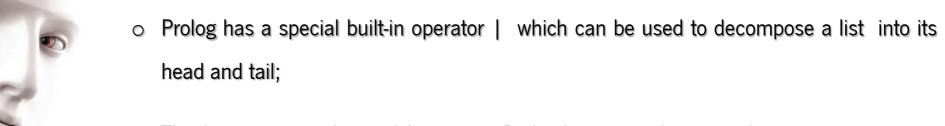
Tail: [ ]

The empty list has neither a head nor a tail;

- For Prolog, [] is a special simple list without any internal structure;
- The empty list plays an important role in recursive predicates for list processing in Prolog.







The | operator is a key tool for writing Prolog list manipulation predicates.

```
?- [Head | Tail] = [ana, julia, miguel, patricia].
```

Head = ana Tail = [julia,miguel,patricia] yes







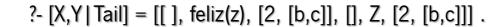
?- [X|Y] = [ana, julia, miguel, patricia].

X = ana

Y = [julia,miguel,patricia]

yes

?\_



$$X = []$$

Y = feliz(z)

Tail = [[2, [b,c]], [], Z, [2, [b,c]]]



### Anonymous variable



```
?- [X1,X2,X3,X4|Tail] = [mara, ana, julia, joana, marco].
```

X1 = mara

X2 = ana

X3 = julia

X4 = joana

Tail = [marco]

yes

?\_



### Anonymous variable



?- [\_,X2,\_,X4|\_] = [mara, ana, julia, joana, marco].

X2 = ana

X4 = joana

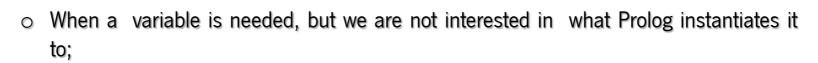
yes

?\_

- Only the 2nd and 4th element of the list;
- \_ indicates anonymous variable.

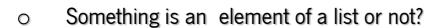






 Each occurrence of the anonymous variable is independent, i.e. can be bound to something different.





O Given a term X and a list L, tells us whether or not X belongs to L

o member/2







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

?- member(ana,[joana,tania,ana,julia]).

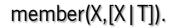
yes

?-

?- member(marco,[joana,tania,ana,julia]).

no





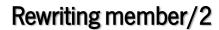
member(X,[H | T]):-member(X,T).

?- member(X,[ana,marco,paulo,julia]).

X = ana;

yes



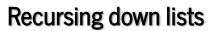






 $member(X,[X | \_]).$ 

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







- member/2 predicate works by recursively working its way down a list;
- doing something to the head, and then;
- recursively doing the same thing to the tail.

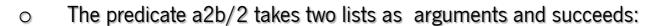
This technique is very common in Prolog.





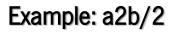
Synthetic Intelligence Lab

Example: a2b/2



- if the first argument is a list of a's, and
- the second argument is a list of b's of exactly the same length.









?- a2b([a,a,a,a],[b,b,b,b]).

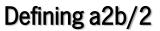
yes

?- a2b([a,a,a,a],[b,b,b]).

no

?- a2b([a,c,a,a],[b,b,b,t]).

no







a2b([],[]).

a2b([a | L1],[b | L2]):-a2b(L1,L2).

?- a2b([a,a,a],[b,b,b]).

yes

?-

?- a2b([a,a,a],[b,c,b]).

no





a2b([],[]).

a2b([a | L1],[b | L2]):-a2b(L1,L2).

?- a2b([a,a,a,a,a], X).

X = [b,b,b,b,b]

yes





- How long is a list?
  - The empty list has length: zero;
  - A non-empty list has length: one plus length of its tail.



### Length of a list in Prolog



```
len([],0).
len([_|L],N):-
len(L,X),
N is X +1.
```

```
?- len([a,b,c,d,e,[a,x],t],X).
X=7
```

yes





- o The predicate acclen/3 has three arguments:
  - list whose length we want to find;
  - length of the list, an integer;
  - An accumulator, keeping track of the intermediate values for the length.



### Length of a list in Prolog

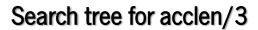


acclen([],Acc,Acc).

?-acclen([a,b,c],0,Len).

Len=3

yes





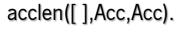
```
acclen([],Acc,Acc).
acclen([_|L],OldAcc,Length):- NewAcc is OldAcc + 1, acclen(L,NewAcc,Length).
       ?- acclen([a,b,c],0,Len).
        / no
                 ?- acclen([b,c],1,Len).
                         ?- acclen([c],2,Len).
              no
                                         ?- acclen([],3,Len).
                         no
```

Len=3

no



### Adding a wrapper predicate

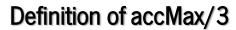


length(List,Length):- acclen(List,0,Length).

?-length([a,b,c], X).

X=3

yes







accMax([H|T],A,Max):- H > A,accMax(T,H,Max).

accMax([H|T],A,Max):- H = < A,accMax(T,A,Max).

accMax([],A,A).

?- accMax([1,0,5,4],0,Max).

Max=5

yes







accMax([H|T],A,Max):- H > A,accMax(T,H,Max).

accMax([H|T],A,Max):- H = < A,accMax(T,A,Max).

accMax([],A,A).

max([H|T],Max):-accMax(T,H,Max).





- o append/3 (whose arguments are all lists)
- Declaratively:
  - append(L1,L2,L3) is true if list L3 is the result of concatenating the lists L1 and L2 together.



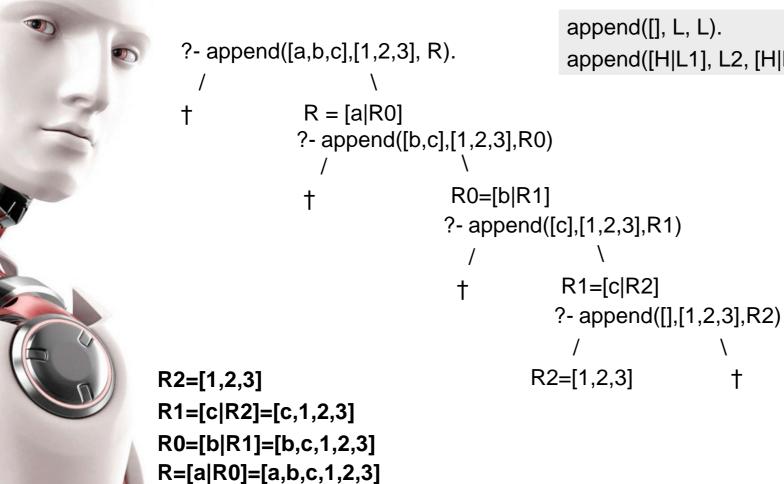
append([], L, L).

append([H|L1], L2, [H|L3]):- append(L1, L2, L3).

- Recursive definition:
  - Base clause: appending the empty list to any list produces that same list;
  - When concatenating a non-empty list [H | T] with a list L, the result is a list with head H and the result of concatenating T and L



## Search tree example



append([J, L, L). append([H|L1], L2, [H|L3]):-append(L1, L2, L3).





Splitting up a list:

?- append(X,Y, [a,b,c,d]).

X=[] Y=[a,b,c,d];

X=[a] Y=[b,c,d];

X=[a,b] Y=[c,d];

X=[a,b,c] Y=[d];

X=[a,b,c,d] Y=[];

no





prefix(P,L):- append(P,\_,L).

- A list P is a prefix of some list L:
  - o there is some list such that L is the result of concatenating P with that list.
- Note the use of the anonymous variable.





prefix(P,L):- append(P,\_,L).

?- prefix(X, [a,b,c,d]).

X=[];

X=[a];

X=[a,b];

X=[a,b,c];

X=[a,b,c,d];

no





suffix(S,L):-append(\_,S,L).

- A list S is a suffix of some list L:
- there is some list such that L is the result of concatenating that list with S.
- Again, the anonymous variable.





suffix(S,L):-append(\_,S,L).

?- suffix(X, [a,b,c,d]). X=[a,b,c,d];

X=[b,c,d];

X=[c,d];

X=[d];

X=[];

no





```
sublist(Sub,List):-
suffix(Suffix,List),
prefix(Sub,Suffix).
```

The sub-lists of a list L are simply the prefixes of suffixes of L



- append/3 can be source of inefficiency:
  - Concatenating a list is not done in one simple action;
  - But by traversing down one of the lists.





reverse([],[]).
reverse([H|T],R):- reverse(T,RT), append(RT,[H],R).

- This definition is correct, but it does an awful lot of work
- It spends a lot of time carrying out appends
- But there is a better way...



#### Reverse using an accumulator



- The better way is using an accumulator;
- The accumulator will be a list, and when start reversing it will be empty;
- Take the head of the list to reverse and add it to the head of the accumulator list;
- Continue this until reaching the empty list;
- At this point the accumulator will contain the reversed list!



#### Reverse using an accumulator



- o accReverse([],L,L).
- accReverse([H|T],Acc,Rev):- accReverse(T,[H|Acc],Rev).





accReverse([],L,L).
accReverse([H|T],Acc,Rev):- accReverse(T,[H|Acc],Rev).

reverse(L1,L2):- accReverse(L1,[],L2).





- List: [a,b,c,d]
- List: [b,c,d]
- List: [c,d]
- List: [d]
- List: []

- Accumulator: []
- Accumulator: [a]
- Accumulator: [b,a]
- Accumulator: [c,b,a]
- Accumulator: [d,c,b,a]



Universidade do Minho
Escola de Engenharia
Departamento de Informática

## Prolog Lists

Mestrado Integrado em Engenharia Informática Licenciatura em Engenharia Informática Inteligência Artificial