Lecture 4: Lists

- Theory
  - Introduce lists, an important recursive data structure often used in Prolog programming
  - Define the member/2 predicate, a fundamental Prolog tool for manipulating lists
  - Illustrate the idea of recursing down lists
  - Define append/3 predicate
  - Discuss reversing a list

Lists

- A list is a finite sequence of elements
- Examples of lists in Prolog:

[mia, vincent, jules, yolanda]
[mia, robber(honeybunny), X, 2, mia]
[ ]
[mia, [vincent, jules], [butch, friend(butch)]]
[[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

Important things about lists

- List elements are enclosed in square brackets
- The length of a list is the number of elements it has
- All sorts of Prolog terms can be elements of a list
- There is a special list:
  the empty list [ ]

Head and Tail example 1

- [mia, vincent, jules, yolanda]

  Head:
  Tail:

Head and Tail example 1

- [mia, vincent, jules, yolanda]

  Head: mia
  Tail:

Head and Tail example 1

- [mia, vincent, jules, yolanda]

  Head: mia
  Tail: [vincent, jules, yolanda]

Head and tail of empty list

- 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 built-in operator |

- Prolog has a special built-in operator | which can be used to decompose a list into its head and tail
- The | operator is a key tool for writing Prolog list manipulation predicates

The built-in operator |

```
?- [X|Y] = [mia, vincent, jules, yolanda].

X = mia
Y = [vincent,jules,yolanda]
yes

?-
```

The built-in operator |

```
?- [X|Y] = [ ].

no

?-
```

The built-in operator |

```
?- [X,Y|Tail] = [[ ], dead(z), [2, [b,c]], [], Z, [2, [b,c]]] .

X = [ ]
Y = dead(z)
Z = _4543
Tail = [[2, [b,c]], [ ], Z, [2, [b,c]]]
yes

?-
```

Anonymous variable

- Suppose we are interested in the second and fourth element of a list

```
?- [X1,X2,X3,X4|Tail] = [mia, vincent, marsellus, jody, yolanda].
X1 = mia
X2 = vincent
X3 = marsellus
X4 = jody
Tail = [yolanda]
yes

?-
```

Anonymous variables

- There is a simpler way of obtaining only the information we want:

```
?- [ _,X2, _,X4|_ ] = [mia, vincent, marsellus, jody, yolanda].
X2 = vincent
X4 = jody
yes

?-
```

- The underscore is the anonymous variable

The anonymous variable

- Is used when you need to use a variable, but you are not interested in what Prolog instantiates it to
- Each occurrence of the anonymous variable is independent, i.e. can be bound to something different

member/2

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

```
?-
```

member/2

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

```
?- member(X,[yolanda,trudy,vincent,jules]).
```

member/2

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

```
?- member(X,[yolanda,trudy,vincent,jules]).
X = yolanda;
X = trudy;
X = vincent;
X = jules;
no
```

Rewriting member/2

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

Recursing down lists

- The 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 and therefore very important that you master it
- So let`s look at another example!

Example: a2b/2

- The predicate a2b/2 takes two lists as arguments and succeeds
  - if the first argument is a list of as, and
  - the second argument is a list of bs 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
```

Defining a2b/2: step 1

```
a2b([],[]).
```

- Often the best away to solve such problems is to think about the simplest possible case
- Here it means: the empty list

Defining a2b/2: step 2

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

- Now think recursively!
- When should a2b/2 decide that two non-empty lists are a list of as and a list of bs of exactly the same length?

Testing a2b/2

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

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

Testing a2b/2

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

```
?- a2b([a,a,a,a],[b,b,b]).
no
?-
```

Testing a2b/2

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

```
?- a2b([a,t,a,a],[b,b,b,c]).
no
?-
```

Further investigating a2b/2

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

```
?- a2b([a,a,a,a,a], X).
X = [b,b,b,b,b]
yes
?-
```

Further investigating a2b/2

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

```
?- a2b(X,[b,b,b,b,b,b,b]).
X = [a,a,a,a,a,a,a]
yes
?-
```

Append

- We will define an important predicate **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([a,b,c,d],[3,4,5],[a,b,c,d,3,4,5]).
yes

?- append([a,b,c],[3,4,5],[a,b,c,d,3,4,5]).
no
```

Append viewed procedurally

- From a procedural perspective, the most obvious use of append/3 is to concatenate two lists together
- We can do this simply by using a variable as third argument

```
?- append([a,b,c,d],[1,2,3,4,5], X).
X=[a,b,c,d,1,2,3,4,5]
yes

?-
```

Definition of append/3

```
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
  - The recursive step says that 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

How append/3 works

- Two ways to find out:
  - Use trace/0 on some examples
  - Draw a search tree!
    Let us consider a simple example

```
?- append([a,b,c],[1,2,3], R).
```

Search tree example

?- append([a,b,c],[1,2,3], R).

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

Search tree example

?- append([a,b,c],[1,2,3], R).

/ \

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

Search tree example

?- append([a,b,c],[1,2,3], R).

/ \

† R = [a|L0]

?- append([b,c],[1,2,3],L0)

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

Search tree example

?- append([a,b,c],[1,2,3], R).

/ \

† R = [a|L0]

?- append([b,c],[1,2,3],L0)

/ \

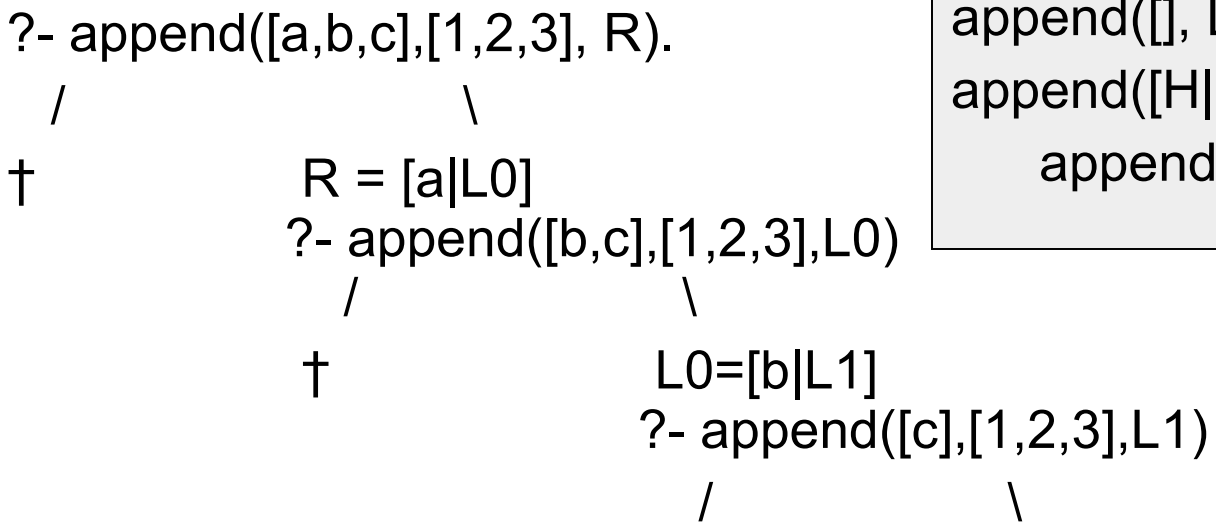
```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

Search tree example

```
?- append([a,b,c],[1,2,3], R).
   /                        \
†                  R = [a|L0]
                   ?- append([b,c],[1,2,3],L0)
                      /                    \
                     †                  L0=[b|L1]
                                        ?- append([c],[1,2,3],L1)
```

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

Search tree example



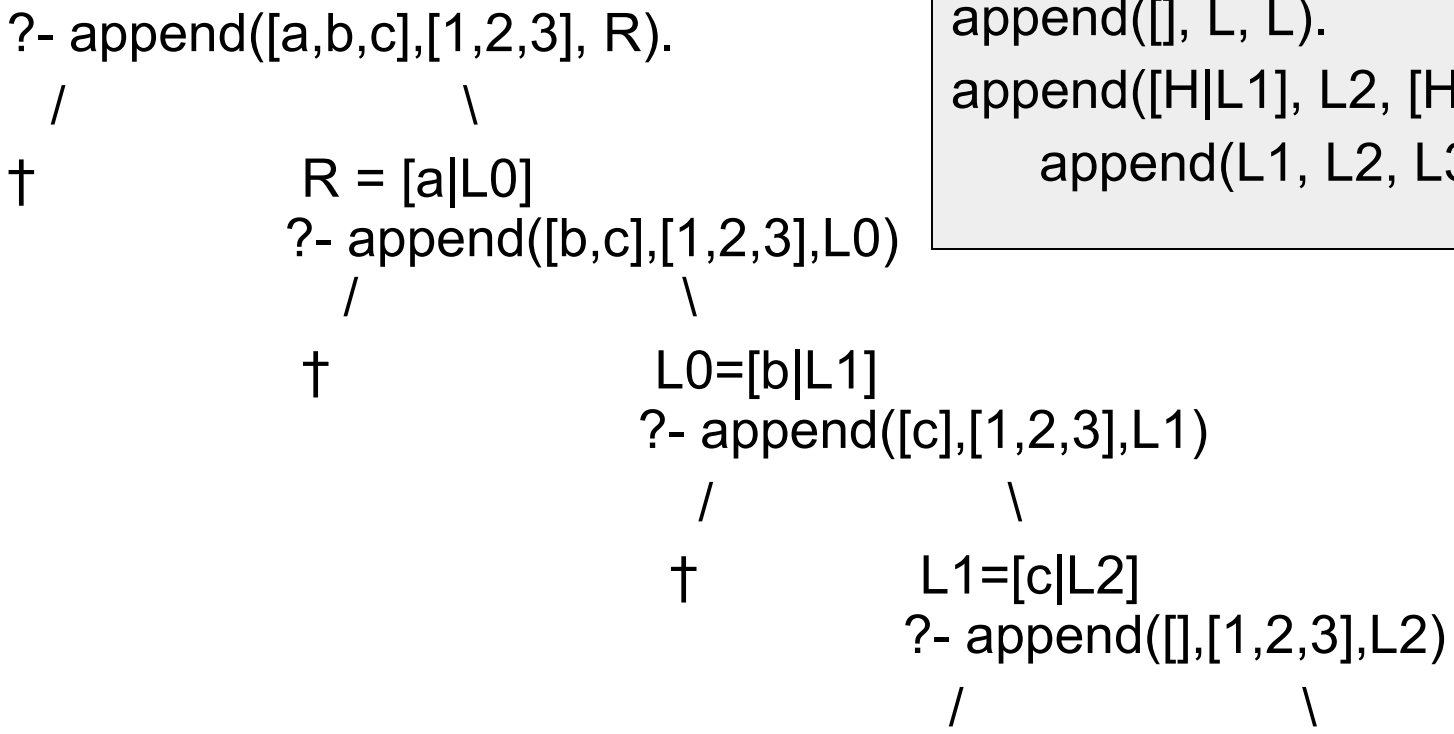

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

Search tree example

```
?- append([a,b,c],[1,2,3], R).
   /                    \
†                R = [a|L0]
                 ?- append([b,c],[1,2,3],L0)
                    /                    \
                   †                  L0=[b|L1]
                                      ?- append([c],[1,2,3],L1)
                                         /                 \
                                        †            L1=[c|L2]
                                                     ?- append([],[1,2,3],L2)
```

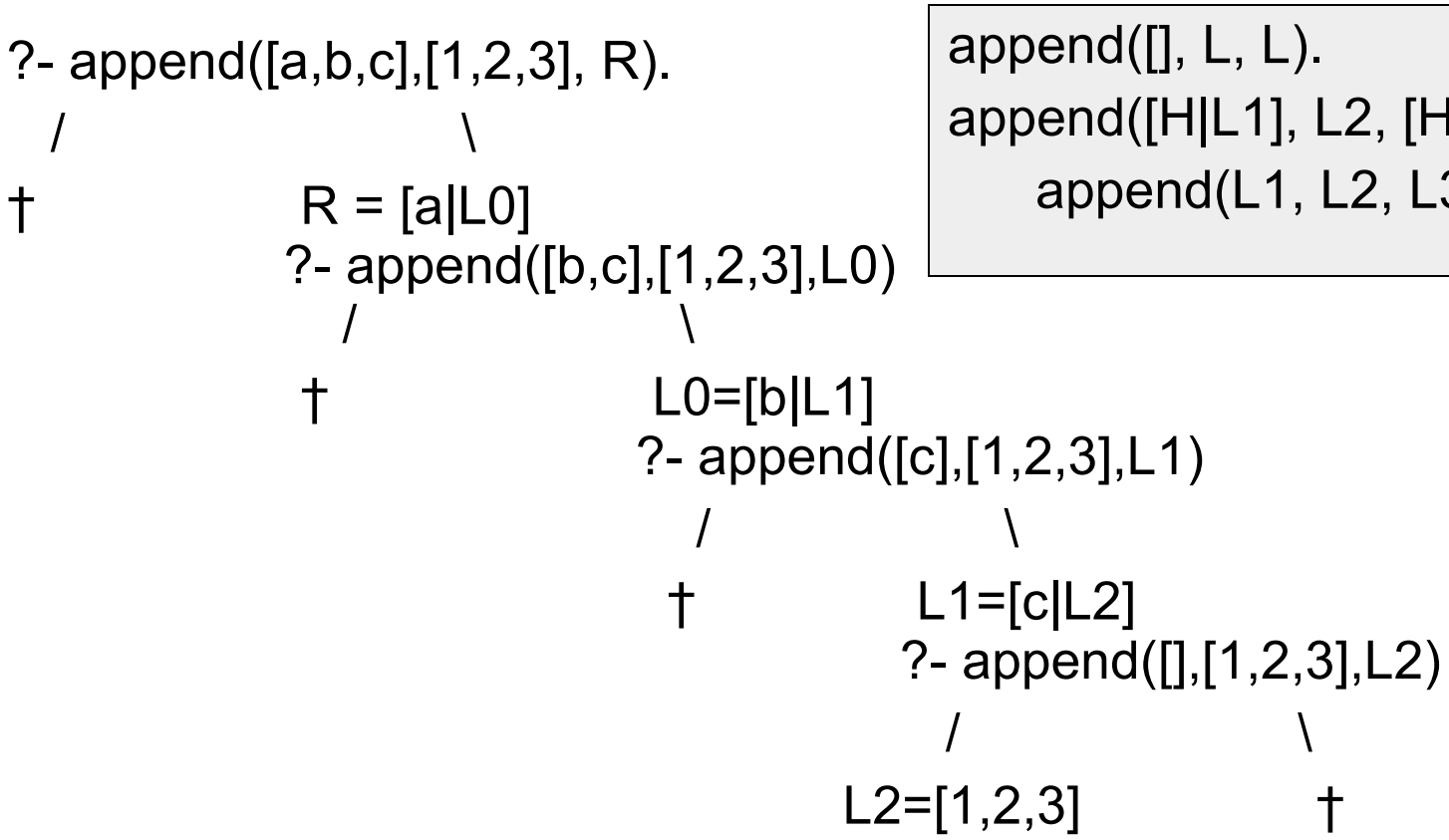
```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

Search tree example



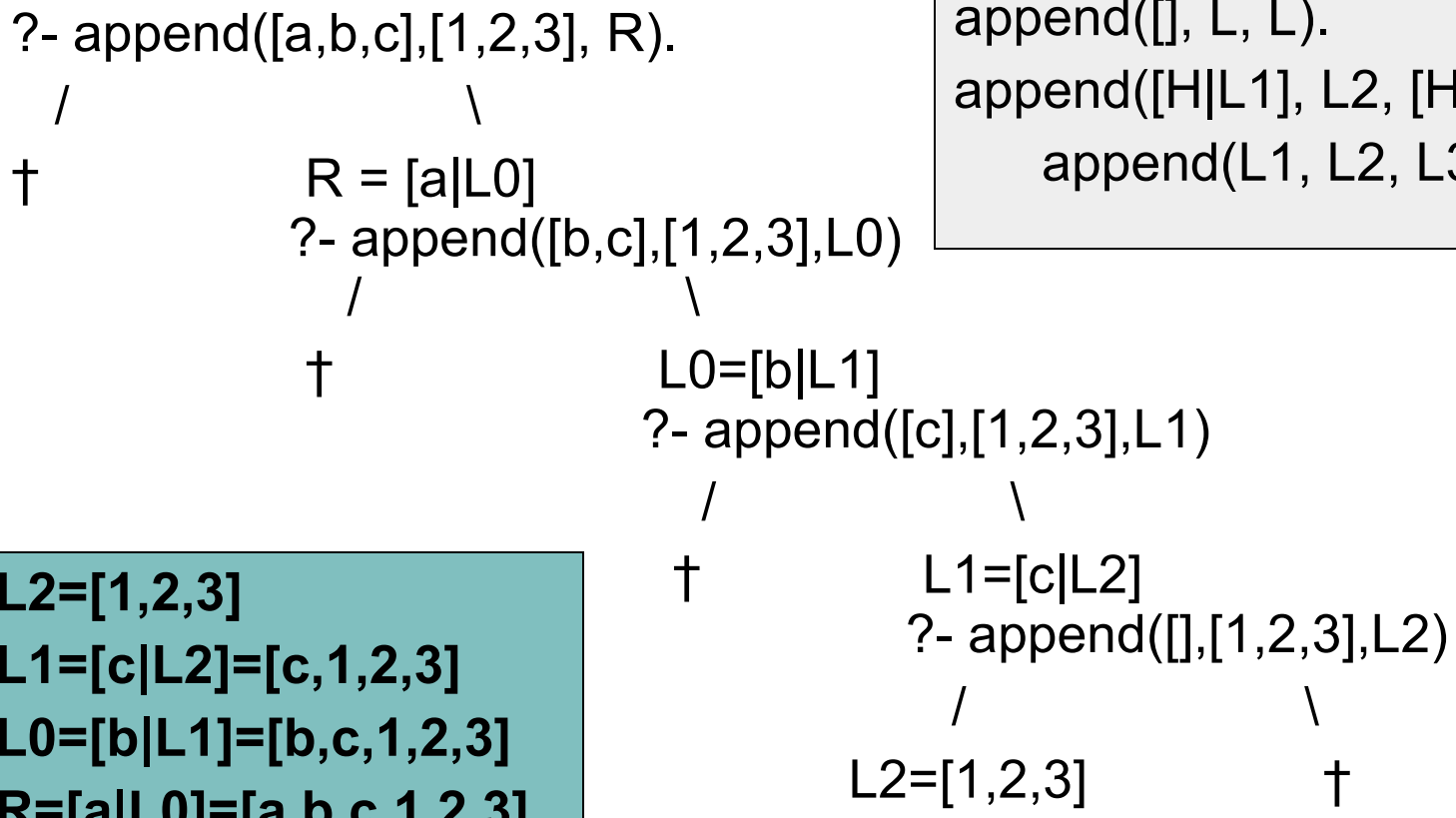

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

Search tree example




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

Search tree example




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

L2=[1,2,3]
L1=[c|L2]=[c,1,2,3]
L0=[b|L1]=[b,c,1,2,3]
R=[a|L0]=[a,b,c,1,2,3]

Using append/3

- Now that we understand how append/3 works, let`s look at some applications
- 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
```

append/3 and efficiency

- The **append/3** predicate is useful, and it is important to know how to to use it
- It is of equal importance to know that **append/3** can be source of inefficiency
- Why?
  - Concatenating a list is not done in a simple action
  - But by traversing down one of the lists

Question

- Using **append/3** we would like to concatenate two lists:
  - List 1: [a,b,c,d,e,f,g,h,i]
  - List 2: [j,k,l]
- The result should be a list with all the elements of list 1 and 2, the order of the elements is not important
- Which of the following goals is the most efficient way to concatenate the lists?

  ?- append([a,b,c,d,e,f,g,h,i],[j,k,l],R).

  ?- append([j,k,l],[a,b,c,d,e,f,g,h,i],R).

Answer

- Look at the way **append/3** is defined
- It recurses on the first argument, not really touching the second argument
- That means it is best to call it with the shortest list as first argument
- Of course you don't always know what the shortest list is, and you can only do this when you don't care about the order of the elements in the concatenated list
- But if you do it can help make your Prolog code more efficient

Reversing a List

- We will illustrate the problem with append/3 by using it to reverse the elements of a list
- That is we will define a predicate that changes a list [a,b,c,d,e] into a list [e,d,c,b,a]
- This would be a useful tool to have, as Prolog only allows easy access to the front of the list

Naïve reverse in Prolog

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

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

Adding a wrapper predicate

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

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

Exercises (1)

Suppose we are given a knowledge base with the following facts:

```
tran(eins,one).
tran(zwei,two).
tran(drei,three).
tran(vier,four).
tran(fuenf,five).
tran(sechs,six).
tran(sieben,seven).
tran(acht,eight).
tran(neun,nine).
```

Write a predicate listtran(G,E) which translates a list of German number words to the corresponding list of English number words. For example:

```
?- listtran([eins,neun,zwei],X).
X = [one,nine,two].
```

Your program should also work in the other direction:

```
?- listtran(X,[one,seven,six,two]).
X = [eins,sieben,sechs,zwei].
```

Exercises (2)

Write a predicate twice(In,Out) whose left argument is a list, and whose right argument is a list consisting of every element in the left list written twice. For example, the query

twice([a,4,buggle],X).

should return

X = [a,a,4,4,buggle,buggle]).

And the query

twice([1,2,1,1],X).

should return

X = [1,1,2,2,1,1,1,1].

Exercises (3)

A palindrome is a word or phrase that spells the same forwards and backwards. For example, `rotator', `eve', and `nurses run' are all palindromes. Write a predicate palindrome(List), which checks whether List is a palindrome. For example, to the queries

?- palindrome([r,o,t,a,t,o,r]).

and

?- palindrome([n,u,r,s,e,s,r,u,n]).

Prolog should respond `yes', but to the query

?- palindrome([n,o,t,h,i,s]).

Prolog should respond `no'.

Exercises (4)

There is a street with three neighboring houses that all have a different color. They are red, blue, and green. People of different nationalities live in the different houses and they all have a different pet. Here are some more facts about them:

The Englishman lives in the red house.

The jaguar is the pet of the Spanish family.

The Japanese lives to the right of the snail keeper.

The snail keeper lives to the left of the blue house.

Who keeps the zebra?

Define a predicate zebra/1 that tells you the nationality of the owner of the zebra.

Hint: Think of a representation for the houses and the street. Code the four constraints in Prolog. member and sublist might be useful predicates.