UNIT

Introduction to Prolog

Lecture 3:

Recursion And Lists

Introduction to Prolog

- Teaching Material
  - Learn Prolog Now!

    http://lpn.swi-prolog.org/lpnpage.php?pageid=top
  - SWI Prolog interpreter

    http://www.swi-prolog.org/
  - Knowledge bases code available at http:///l3s.de/~gaugaz/ki/prolog/

Lecture 3: Recursion

- **Theory**
  - Introduce recursive definitions in Prolog
  - Four examples
  - Show that there can be mismatches between the declarative and procedural meaning of a Prolog program

- **Exercises**
  - Corrections exercises LPN chapter 2
  - Exercises of LPN chapter 3

Recursive Definitions

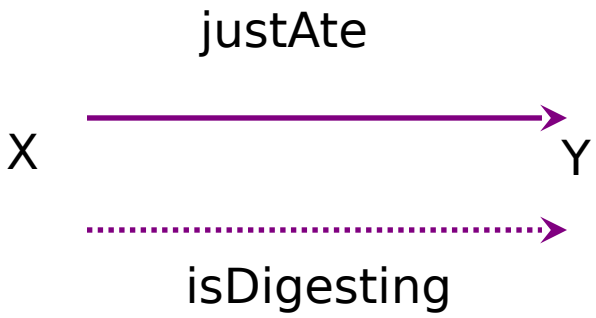
- Prolog predicates can be defined recursively
- A predicate is recursively defined if one or more rules in its definition refers to itself

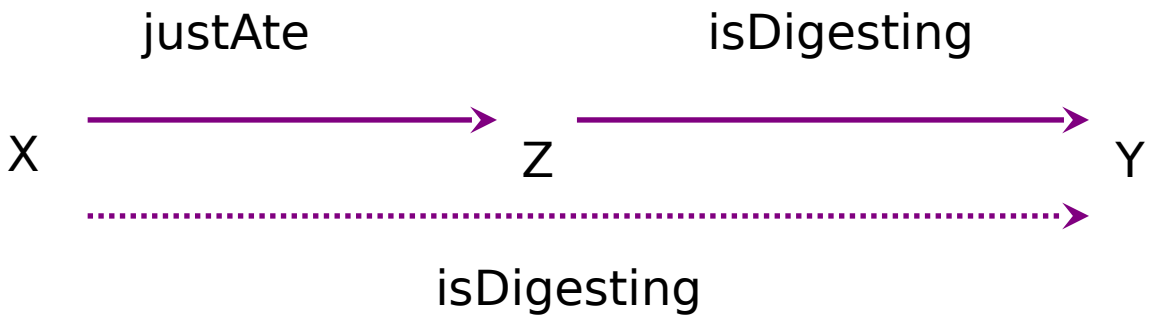
Example 1: Eating

```
isDigesting(X,Y):- justAte(X,Y).
isDigesting(X,Y):- justAte(X,Z), isDigesting(Z,Y).

justAte(mosquito,blood(john)).
justAte(frog,mosquito).
justAte(stork,frog).
```

```
?-
```

Picture of the situation





Example 1: Eating


```
isDigesting(X,Y):- justAte(X,Y).
isDigesting(X,Y):- justAte(X,Z), isDigesting(Z,Y).
justAte(mosquito,blood(john)).
justAte(frog,mosquito).
justAte(stork,frog).
```

```
?- isDigesting(stork,mosquito).
true
```

Another recursive definition

```
p:- p.
```

```
?- p.
ERROR: out of memory
```

Example 2: Decendant


```
child(bridget,caroline).
child(caroline,donna).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
```

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
```

Example 2: Decendant


```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
```

Example 2: Decendant


```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).


descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
```

```
?- descend(anna,donna).
no
?-
```

Example 2: Decendant


```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).


descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
descend(X,Y):- child(X,Z), child(Z,U), child(U,Y).
```

```
?-
```

Example 2: Decendant

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).


descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), descend(Z,Y).
```

```
?-
```

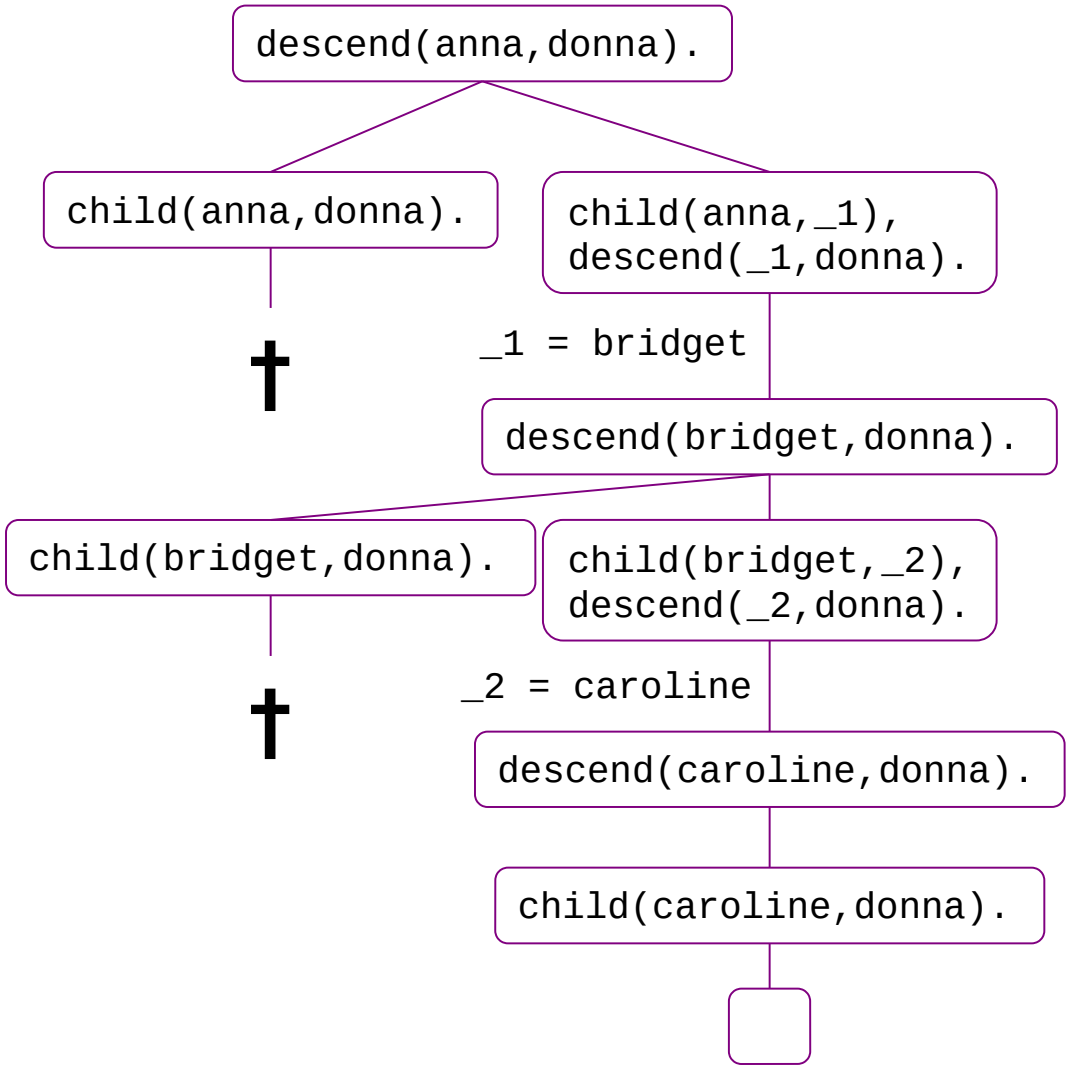
Example 2: Decendant


```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).


descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), descend(Z,Y).
```

```
?- descend(anna,donna).
```

Example 2: Search tree for descend(anna,donna)

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).


descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
     descend(Z,Y).
```



Example 3: Successor

- Suppose we use the following way to write numerals:
  1. 0 is a numeral.
  2. If X is a numeral, then so is succ(X).

Example 3: Successor

```
numeral(0).
numeral(succ(X)):- numeral(X).
```

Example 3: Successor


```
numeral(0).
numeral(succ(X)):- numeral(X).
```

```
?- numeral(succ(succ(succ(0)))).
yes
?-
```

Example 3: Successor


```
numeral(0).
numeral(succ(X)):- numeral(X).
```

```
?- numeral(X).
```

Example 3: Successor

```
numeral(0).
numeral(succ(X)):- numeral(X).
```

```
?- numeral(X).
X=0;
X=succ(0);
X=succ(succ(0));
X=succ(succ(succ(0)));
X=succ(succ(succ(succ(0))))
```

Example 4: Addition

```
?- add(succ(succ(0)),succ(succ(succ(0))), Result).
Result=succ(succ(succ(succ(succ(0)))))
yes
```

Example 4: Addition


```
add(0,X,X).                          %%% base clause
```

```
?- add(succ(succ(0)),succ(succ(succ(0))), Result).
Result=succ(succ(succ(succ(succ(0)))))
yes
```

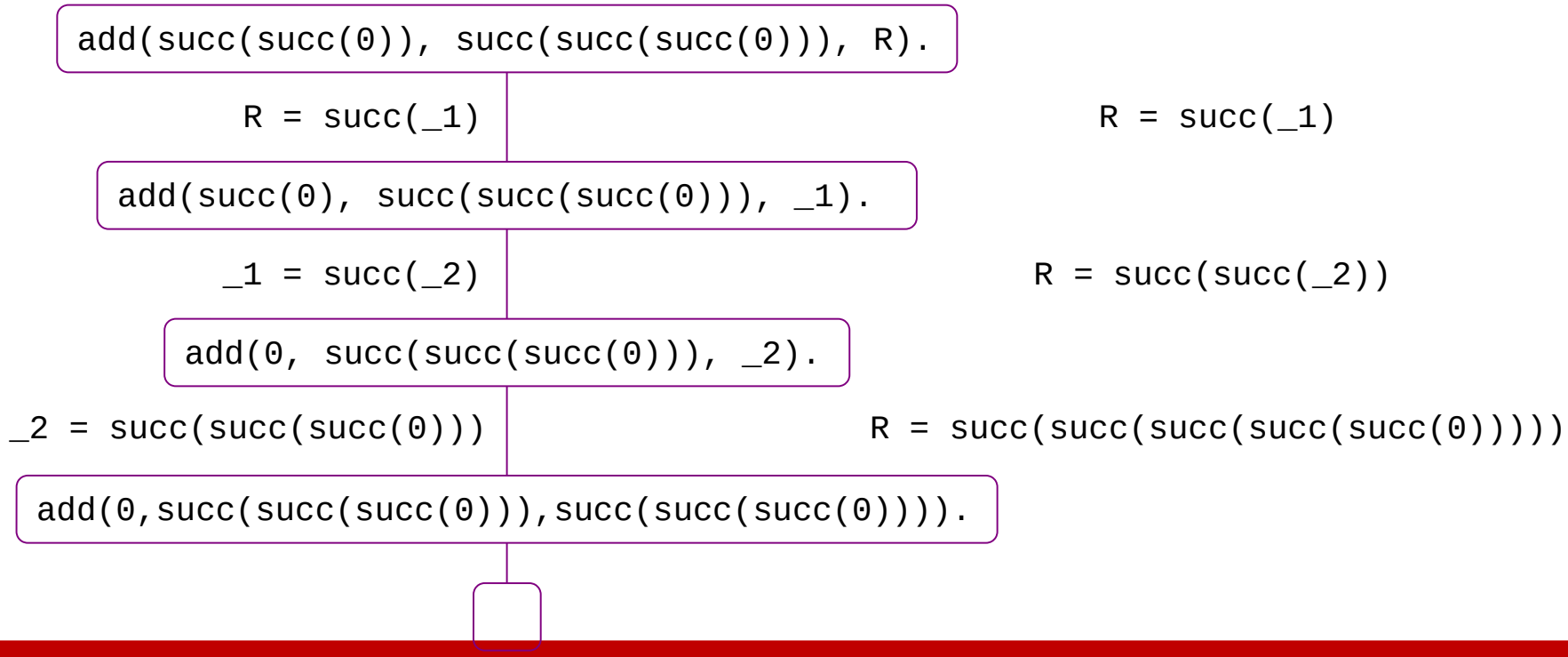
Example 4: Addition


```
add(0,X,X).                          %%% base clause

add(succ(X),Y,succ(Z)):-            %%% recursive clause
      add(X,Y,Z).
```

```
?- add(succ(succ(0)),succ(succ(succ(0))), Result).
Result=succ(succ(succ(succ(succ(0)))))
yes
```

Example 4: Search tree


```
add(0,X,X).                          %%% base clause

add(succ(X),Y,succ(Z)):-             %%% recursive clause
    add(X,Y,Z).
```



- Call: (6) add(succ(succ(succ(0))), succ(succ(0)), R)
- Call: (7) add(succ(succ(0)), succ(succ(0)), _G648)
- Call: (8) add(succ(0), succ(succ(0)), _G650)
- Call: (9) add(0, succ(succ(0)), _G652)
- Exit: (9) add(0, succ(succ(0)), succ(succ(0)))
- Exit: (8) add(succ(0), succ(succ(0)), succ(succ(succ(0))))
- Exit: (7) add(succ(succ(0)), succ(succ(0)), succ(succ(succ(succ(0)))))
- Exit: (6) add(succ(succ(succ(0))), succ(succ(0)), succ(succ(succ(succ(succ(0))))))

Prolog and Logic

- Prolog was the first reasonable attempt to create a logic programming language
  - Programmer gives a declarative specification of the problem, using the language of logic
  - The programmer should not have to tell the computer what to do
  - To get information, the programmer simply asks a query
- Prolog does some important steps in this direction, but nevertheless,
  Prolog is not a full logic programming language!
- Prolog has a specific way of answering queries:
  - Search knowledge base from top to bottom
  - Processes clauses from left to right
  - Backtracking to recover from bad choices

descend1.pl

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), descend(Z,Y).
```

```
?- descend(A,B).
A=anna
B=bridget
```

- **A** = anne,
  B = bridget
- **A** = bridget,
  B = caroline
- **A** = caroline,
  B = donna
- **A** = donna,
  B = emily
- **A** = anne,
  B = caroline
- **A** = anne,
  B = donna
- **A** = anne,
  B = emily
- **A** = bridget,
  B = donna
- **A** = bridget,
  B = emily
- **A** = caroline,
  B = emily
- **false**

descend2.pl

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Z), descend(Z,Y).
descend(X,Y):- child(X,Y).
```

```
?- descend(A,B).
A=anna
B=emily
```

- **A** = anne,
  B = emily
- **A** = anne,
  B = donna
- **A** = anne,
  B = caroline
- **A** = bridget,
  B = emily
- **A** = bridget,
  B = donna
- **A** = caroline,
  B = emily
- **A** = anne,
  B = bridget
- **A** = bridget,
  B = caroline
- **A** = caroline,
  B = donna
- **A** = donna,
  B = emily

descend3.pl

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- descend(Z,Y), child(X,Z).
descend(X,Y):- child(X,Y).
```

```
?- descend(A,B).
ERROR: OUT OF LOCAL STACK
```

descend4.pl

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Y).
descend(X,Y):- descend(Z,Y), child(X,Z).
```

```
?- descend(A,B).
```

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

- A non-empty list can be thought of as consisting of two parts
  - The head
  - The tail
- The head is the first item in the list
- The tail is everything else
  - The tail is the list that remains when we take the first element away
  - The tail of a list is always a list

Head and Tail examples

- [mia, vincent, jules, yolanda]

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

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

  Head: [ ]
  Tail: [dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

- [dead(z)]

  Head: dead(z)
  Tail: [ ]

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 |

1

```
?- [Head|Tail] = [mia, vincent, jules, yolanda].

Head = mia
Tail = [vincent,jules,yolanda]
yes

?-
```

2

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

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

?-
```

3

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

no

?-
```

The built-in operator |


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

X = [ ]
Y = dead(Z)
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 (of a List)

- One of the most basic things we would like to know is whether something is an element of a list or not
- So let`s write a predicate that when given a term X and a list L, tells us whether or not X belongs to L
- This predicate is usually called 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(yolanda,[yolanda,trudy,vincent,jules]).
```

member/2

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

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

member/2

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

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

member/2

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

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

member/2

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

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

member/2

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

```
?- member(zed,[yolanda,trudy,vincent,jules]).
no
?-
```

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).
```

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

Exercises

Learn Prolog Now! Chapter 3,4

by Patrick Blackburn, Johan Bos, and Kristina Striegnitz

[next] [prev] [prev-tail] [tail] [up]

Chapter 1
Facts, Rules, and Queries

This chapter has two main goals:

1. To give some simple examples of Prolog programs. This will introduce us to the three basic constructs in Prolog: facts, rules, and queries. It will also introduce us to a number of other themes, like the role of logic in Prolog, and the idea of performing unification with the aid of variables.
2. To begin the systematic study of Prolog by defining terms, atoms, variables and other syntactic concepts.

[next] [prev] [prev-tail] [front] [up]

Summary of this lecture

- In this lecture we introduced recursive predicates
- We also looked at the differences between the declarative and the procedural meaning of Prolog programs
- We have identified some of the shortcomings of Prolog seen as a logical programming language
- In this lecture we introduced list and recursive predicates that work on lists
- The kind of programming that these predicates illustrated is fundamental to Prolog
- You will see that most Predicates you will write in your Prolog career will be variants of these predicates