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

- Exercises
  - Solution exercises LPN chapter 3
  - Exercises of LPN chapter 4

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
  - <u>The tail of a list is always a list</u>

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 example 2

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

  Head:
  Tail:

Head and Tail example 2

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

  Head: [ ]
  Tail:

Head and Tail example 2

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

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

Head and Tail example 3

- [dead(z)]

  Head:
  Tail:

Head and Tail example 3

- [dead(z)]

  Head: dead(z)
  Tail:

Head and Tail example 3

- [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 |

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

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


?-
```

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 (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/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).
```

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
?-
```

Summary of this lecture

- 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

Solution Exercise 3.2

Define a predicate greater_than/2 that takes two numerals in the notation that we introduced in this lecture (i.e. 0, succ (0), succ(succ(0)) ...) as arguments and decides whether the first one is greater than the second one. E.g:

```
?- greater_than(succ(succ(succ(0))),succ(0)).
   yes
?- greater_than(succ(succ(0)),succ(succ(succ(0)))).
   no
```

Solution Exercise 3.2

```
greater_than(succ(A),0)).
greater_than(succ(A),succ(B)) :- greater_than(A,B).
```

```
?- greater_than(succ(succ(succ(0))),succ(0)).
   yes
?- greater_than(succ(succ(0)),succ(succ(succ(0)))).
   no
```

Solution Exercise 3.4

In the lecture, we saw the predicate

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

Could we have formulated this predicate as follows?

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

Compare the declarative and the procedural meaning of this predicate definition.

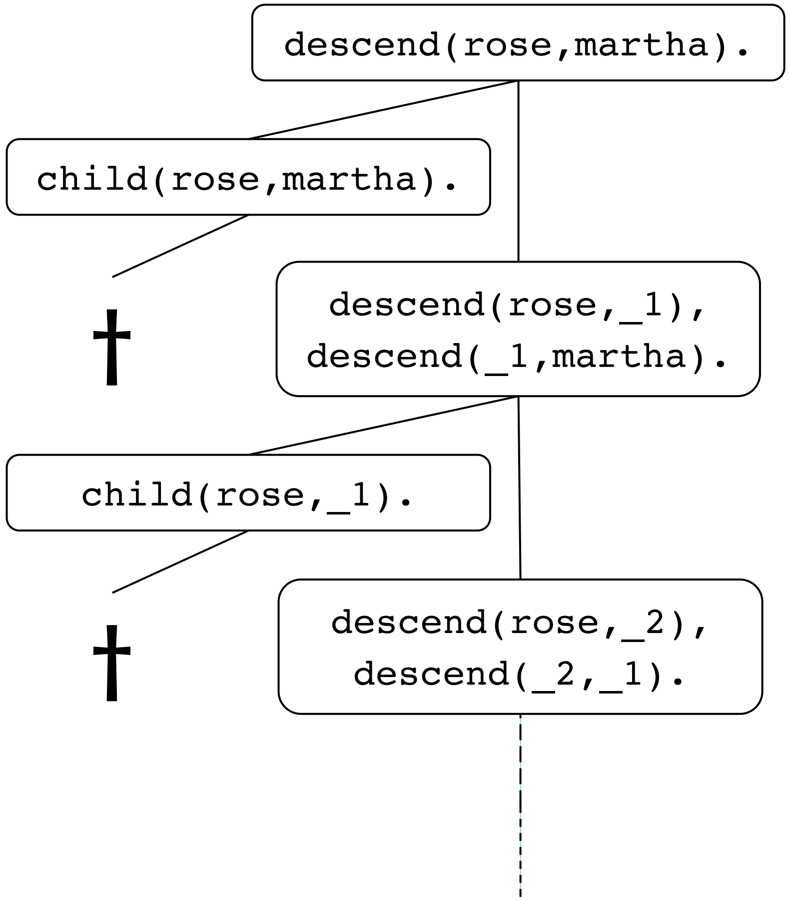
Hint: What happens when you ask the query descend (rose,martha)?

Beware: Some examples are different in the slides than in the online LPN book! Ex. 3.4 refers to the KB in the book!

Solution Exercise 3.4

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

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



Infinite loop if descend(X,Z) cannot be proved true by child (X,Z)!

Exercises LPN Chapter 4

- 4.1, 4.3, 4.4

Next lecture

- Prolog lecture next Thursday May 5 cancelled
  - KI lecture in the afternoon takes place!
- May 12: Introduce **arithmetic** in Prolog
  - Introduce Prolog`s built-in abilities for performing arithmetic
  - Apply them to simple list processing problems
  - Introduce the idea of accumulators