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
  – Exercises of LPN chapter 4
  – Practical work
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 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

- \([\;\;], \text{dead}(z), [2, [b,c]], [\;\;], Z, [2, [b,c]]\)

Head:  
Tail:
Head and Tail example 2

- \([ [ ], \text{dead}(z), [2, [b,c]], [ ], Z, [2, [b,c]]] \)

Head: \([ ]\)
Tail:
Head and Tail example 2

- $[[\,], \text{dead}(z), [2, [b,c]], [\,], Z, [2, [b,c]]]$  

  Head: $[\,]$  
  Tail: $[	ext{dead}(z), [2, [b,c]], [\,], Z, [2, [b,c]]]$
Head and Tail example 3

\[ \text{Head: } [\text{dead}(z)] \]

\[ \text{Tail: } \]

Head and Tail
Head and Tail example 3

• \[\text{dead}(z)\]

Head: \text{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 $\mid$

- Prolog has a special built-in operator $\mid$ which can be used to decompose a list into its head and tail.
- The $\mid$ 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 |

?- \([X|Y] = [\text{mia, vincent, jules, yolanda}]\).

\[X = \text{mia}\]
\[Y = [\text{vincent,jules,yolanda}]\]
yes

??
The built-in operator $\mathcal{O}$
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
Exercises

• Exercise 4.1 of LPN
• Exercise 4.2 of LPN
• 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(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

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

a2b([],[]).
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`

```prolog
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.
Next lecture

• 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