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)
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 an 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

- 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
?-
```

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([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([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).
```

```
?- 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)
                                                     /                    \
```

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)
                                                               /                      \
                                                         L2=[1,2,3]                    †
```

```
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).
```

```
?- 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)
                                                    /                 \
                                               L2=[1,2,3]              †
```

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
```

Prefix and suffix

- We can also use append/3 to define other useful predicates
- A nice example is finding prefixes and suffixes of a list

Definition of prefix/2

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

- A list P is a prefix of some list L when there is some list such that L is the result of concatenating P with that list.
- We use the anonymous variable because we don`t care what that list is.

Use of prefix/2

```
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
```

Definition of suffix/2

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

- A list S is a suffix of some list L when there is some list such that L is the result of concatenating that list with S.
- Once again, we use the anonymous variable because we don`t care what that list is.

Use of suffix/2

```
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
```

Definition of sublist/2

- Now it is very easy to write a predicate that finds sub-lists of lists
- The sub-lists of a list L are simply the prefixes of suffixes of L

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

Reversing a List

- 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

Reverse

- Recursive definition
  1. If we reverse the empty list, we obtain the empty list
  2. If we reverse the list [H|T], we end up with the list obtained by reversing T and concatenating it with [H]
- To see that this definition is correct, consider the list [a,b,c,d].
  - If we reverse the tail of this list we get [d,c,b].
  - Concatenating this with [a] yields [d,c,b,a]

Reverse

```
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

- A better way is using an accumulator
- The accumulator will be a list, and when we start reversing it will be empty
- We simply take the head of the list that we want to reverse and add it to the the head of the accumulator list
- We continue this until we hit the empty list
- At this point the accumulator will contain the reversed list!

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).
```

Illustration of the accumulator

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

Illustration of the accumulator

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

Illustration of the accumulator

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

Illustration of the accumulator

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

Illustration of the accumulator

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

A few other things …

Comparing Integers

Arithmetic	Prolog
x < y	X < Y
x ≤ y	X =< Y
x = y	X =:= Y
x ≠ y	X =\= Y
x ≥≥ y	X >= Y
x > y	X > Y

Comparison Operators

- Have the obvious meaning
- Force both left and right hand argument to be evaluated

```
?- 2 < 4+1.
yes

?- 4+3 > 5+5.
no
```

Comparison Operators

- The unification operator does not force evaluation but the numeric equality comparison operator dose.

```
?- 4 = 4.
yes

?- 2+2 = 4.
no

?- 2+2 =:= 4.
yes
```

Similar looking symbols

= The unification predicate. Succeeds if it can unify its arguments, fails otherwise.

\= The negation of the unification predicate. Succeeds if = fails, and vice-versa.

== The identity predicate. Succeeds if its arguments are identical, fails otherwise.

\== The negation of the identity predicate. Succeeds if == fails, and vice-versa.

=:= The arithmetic equality predicate. Succeeds if its arguments evaluate to the same integer.

=\= The arithmetic inequality predicate. Succeeds if its arguments evaluate to different integers.

Negation as Failure

- The cut operator (!) suppresses backtracking.
- The fail predicate always fails.
- They can be combined to get **negation as failure** as follows:

```
neg(Goal):- Goal, !, fail.
neg(Goal).
```

Vincent and burgers

```
enjoys(vincent,X):- burger(X),
                    neg(bigKahunaBurger(X)).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

Vincent and burgers

```
enjoys(vincent,X):- burger(X),
                    neg(bigKahunaBurger(X)).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

```
?- enjoys(vincent,X).
X=a
X=c
X=d
```

Another built-in predicate: \+

- Because negation as failure is so often used, there is no need to define it.
- In standard Prolog the prefix operator \+ means negation as failure
- We can define Vincent`s preferences as follows:

```
?- enjoys(vincent,X).
X=a
X=c
X=d
```

```
enjoys(vincent,X):-  burger(X),
                     \+ bigKahunaBurger(X).
```

Negation as failure and logic

- Negation as failure is not logical negation
- Changing the order of the goals in the vincent and burgers program gives a different behaviour:

```
enjoys(vincent,X):- \+ bigKahunaBurger(X),
                       burger(X).
```

```
?- enjoys(vincent,X).
no
```

Complete negation example

```
enjoys(vincent,X):- burger(X),
                    \+ bigKahunaBurger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

```
?- enjoys(vincent,X).
X=a;
X=c;
X=d;
no
```

Miscellaneous Examples and Applications

Quick sort in Prolog

```
partition([], Pivot, [], []).
partition([H|T], Pivot, [H|Left], Right) :- H =< Pivot,
                            partition(T, Pivot, Left, Right).
partition([H|T], Pivot, Left, [H|Right]) :- H > Pivot,
                            partition(T, Pivot, Left, Right).

quicksort([], []).
quicksort([P|T], Sorted) :- partition(T, P, Left, Right),
                    quicksort(Left, SortedLeft),
                    quicksort(Right, SortedRight),
                    append(SortedLeft, [P|SortedRight], Sorted).
```

Complex queries and Expert systems

```
person(ann).
person(bob).

personSkill(ann, software).
personSkill(bob, software).
personSkill(bob, floating).

job(lifeguard).
job(developer).

jobSkill(lifeguard, firstaid).
jobSkill(lifeguard, swimming).

hasSkills(P, Skills) :- person(P), findall(S, personSkill(P,S), Skills).
needsSkills(J, Skills) :- job(J), findall(S, jobSkill(J,S), Skills).

fit_for_job(P, J) :- hasSkills(P, SkillsHas),
            needsSkills(J, SkillsNeeded),
            forall(member(S, SkillsNeeded), member(S, SkillsHas)).
```

Natural language processing

- See:
  - Parsing and difference lists
  - Eliza chatbot
- IBM Watson:
  - Unstructured Information Management Architecture framework running on Hadoop

"We required a language in which we could conveniently express pattern matching rules over the parse trees and other annotations (such as named entity recognition results), and a technology that could execute these rules very efficiently. We found that Prolog was the ideal choice for the language due to its simplicity and expressiveness." -- Watson development team