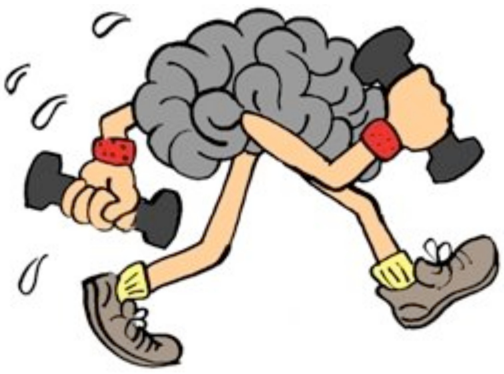
Lecture 6: More Lists

- Theory
  - Define **append/3**, a predicate for concatenating two lists, and illustrate what can be done with it
  - Discuss two ways of **reversing** a list
    - A naïve way using append/3
    - A more efficient method using accumulators

Lecture 6: More Lists

- Exercises
  - Exercises of LPN: 6.1, 6.2, 6.3, 6.4, 6.5, 6.6
  - Practical work



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's 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|R0]
                   ?- append([b,c],[1,2,3],R0)
```

```
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|R0]
                     ?- append([b,c],[1,2,3],R0)
                         /                       \
```

```
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|R0]
                  ?- append([b,c],[1,2,3],R0)
                     /                  \
                  †                   R0=[b|R1]
                                     ?- append([c],[1,2,3],R1)
```

```
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|R0]
                       ?- append([b,c],[1,2,3],R0)
                          /                        \
                       †                          R0=[b|R1]
                                                 ?- append([c],[1,2,3],R1)
                                                     /                     \
```

```
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|R0]
                  ?- append([b,c],[1,2,3],R0)
                     /                    \
                  †                    R0=[b|R1]
                                      ?- append([c],[1,2,3],R1)
                                         /                  \
                                      †              R1=[c|R2]
                                                    ?- append([],[1,2,3],R2)
```

```
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|R0]
                   ?- append([b,c],[1,2,3],R0)
                      /                       \
                   †                     R0=[b|R1]
                                        ?- append([c],[1,2,3],R1)
                                           /                  \
                                        †             R1=[c|R2]
                                                     ?- append([],[1,2,3],R2)
                                                        /                    \
```

```
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|R0]
                     ?- append([b,c],[1,2,3],R0)
                        /                     \
                     †                     R0=[b|R1]
                                          ?- append([c],[1,2,3],R1)
                                             /                  \
                                          †                R1=[c|R2]
                                                          ?- append([],[1,2,3],R2)
                                                             /                    \
                                                      R2=[1,2,3]                  †
```

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|R0]
                    ?- append([b,c],[1,2,3],R0)
                       /                    \
                    †                   R0=[b|R1]
                                       ?- append([c],[1,2,3],R1)
                                          /                  \
                                       †                 R1=[c|R2]
                                                        ?- append([],[1,2,3],R2)
                                                           /                  \
                                                    R2=[1,2,3]                 †
```

R2=[1,2,3]
R1=[c|R2]=[c,1,2,3]
R0=[b|R1]=[b,c,1,2,3]
R=[a|R0]=[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 couldn't care less 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).
```

append/3 and efficiency

- The **append/3** predicate is useful, and it is important to know how 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 one 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 gives easy access to the front of the list

Naïve 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]

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

- The 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 head of the accumulator list
- We continue this until we reach the empty list
- At this point the accumulator will contain the reversed list!

Reverse using an accumulator

- The 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 reach the empty list
- At this point the accumulator will contain the reversed list!

Reverse using an accumulator

- The 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 head of the accumulator list
- We continue this until we reach the empty list
- At this point the accumulator will contain the reversed list!

Reverse using an accumulator

- The 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 head of the accumulator list
- We continue this until we reach the empty list
- At this point the accumulator will contain the reversed list!

Reverse using an accumulator

- The 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 head of the accumulator list
- We continue this until we reach 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]

Summary of this lecture

- The **append/3** is a useful predicate, don't be scared of using it
- However, it can be a source of inefficiency
- The use of accumulators is often better
- We will encounter a very efficient way of concatenating list in later lectures, where we will explore the use of "difference lists"

Next lecture

- Definite Clause Grammars
  - Introduce context free grammars and some related concepts
  - Introduce DCGs, definite clause grammars, a built-in Prolog mechanism for working with context free grammars