Tutorial 7

More Functions with Lists

1. Append

We shall define an important predicate `append/3` whose arguments are all lists. Viewed declaratively, `append(L1,L2,L3)` will hold when the list `L3` is the result of concatenating the lists `L1` and `L2` together. For example, if we pose the query

```
?- append([a,b,c],[1,2,3],[a,b,c,1,2,3]).
```

we will get the response 'yes'. On the other hand, if we pose the query

```
?- append([a,b,c],[1,2,3],[a,b,c,1,2]).
```

we will get the answer 'no'.

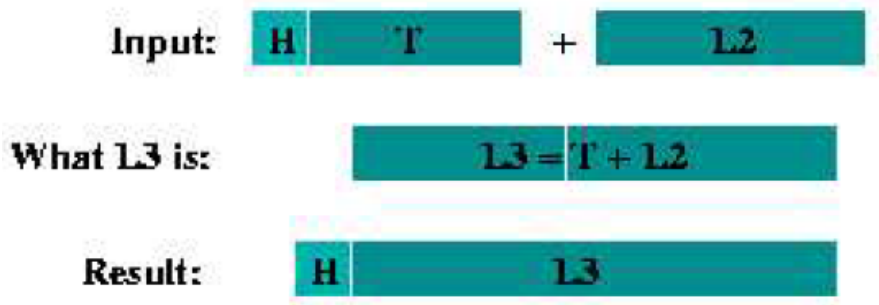
From a procedural perspective, the most obvious use of append is to concatenate two lists together.
But we can also use `append` to split up a list. In fact, `append` is a real workhorse. There's lots we can do with it, and studying it is a good way to gain a better understanding of list processing in Prolog.

1.1. Defining Append

Here's how append/3 is defined:

```
append([],L,L).
append([H|T],L2,[H|L3]) :- append(T,L2,L3).
```

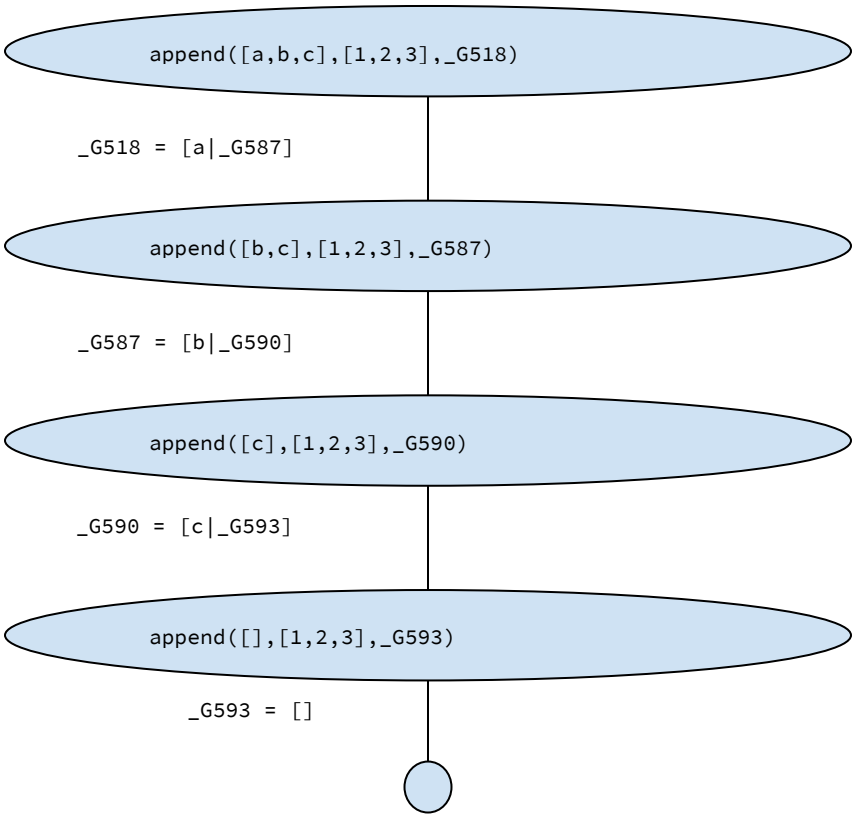
This is a recursive definition. Representing it pictorially:



If we carried out a trace on what happens next, we would get something like the following:

```
append([a, b, c], [1, 2, 3], _G518)
append([b, c], [1, 2, 3], _G587)
append([c], [1, 2, 3], _G590)
append([], [1, 2, 3], _G593)
append([], [1, 2, 3], [1, 2, 3])
append([c], [1, 2, 3], [c, 1, 2, 3])
append([b, c], [1, 2, 3], [b, c, 1, 2, 3])
append([a, b, c], [1, 2, 3], [a, b, c, 1, 2, 3])
X = [a, b, c, 1, 2, 3]
yes
```

The basic pattern should be clear: in the first four lines we see that Prolog recurses its way down the list in its first argument until it can apply the base case of the recursive definition. Then, as the next four lines show, it then stepwise 'fills in' the result. This 'filling in' process is carried out by successively instantiating the variables `_G593`, `_G590`, `_G587`, and `_G518`. Try to relate to the search tree given below:



Work through this example carefully, and make sure you fully understand the pattern of variable
instantiations, namely:

```
_G518 = [a|_G587] = [a|[b|_G590]] = [a|[b|[c|_G593]]]
```

1.2. Using Append

One important use of append is to split up a list into two consecutive lists. For example:

```
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
```

Thus, we can define a program which finds **prefixes** of lists. For example, the prefixes of `[a,b,c,d]` are `[]`, `[a]`, `[a,b]`, `[a,b,c]`, and `[a,b,c,d]`. With the help of append it is straightforward to define a program `prefix/2`, whose arguments are both lists, such that `prefix(P,L)` will hold when `P` is a prefix of `L`. Here's how:

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

This predicate successfully finds prefixes of lists, and moreover, via backtracking, finds them all:

```
prefix(X,[a,b,c,d]).
X = [] ;
X = [a] ;
X = [a,b] ;
```

```
X = [a,b,c] ;
X = [a,b,c,d] ;
no
```

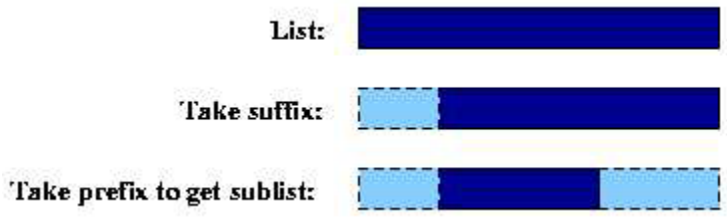
In a similar fashion, we can define a program which finds suffixes of lists. For example, the **suffixes** of `[a,b,c,d]` are `[]`, `[d]`, `[c,d]`, `[b,c,d]`, and `[a,b,c,d]`. Again, using append it is easy to define `suffix/2`, a predicate whose arguments are both lists, such that `suffix(S,L)` will hold when `S` is a suffix of `L`:

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

This predicate successfully finds suffixes of lists, and moreover, via backtracking, finds them all:

```
suffix(X,[a,b,c,d]).
X = [a,b,c,d] ;
X = [b,c,d] ;
X = [c,d] ;
X = [d] ;
X = [] ;
no
```

Now it's very easy to define a program that finds **sublists** of lists. The sublists of `[a,b,c,d]` are `[]`, `[a]`, `[b]`, `[c]`, `[d]`, `[a,b]`, `[b,c]`, `[c,d]`, `[d,e]`, `[a,b,c]`, `[b,c,d]`, and `[a,b,c,d]`. A little thought reveals that the sublists of a list L are simply the *prefixes of suffixes* of `L`. Pictorially representing:



Thus we simply define `sublist/2` as:

```
sublist(SubL,L) :- suffix(S,L),prefix(SubL,S).
```

That is, `SubL` is a sublist of `L` if there is some suffix `S` of `L` of which `SubL` is a prefix. This program doesn't *explicitly* use append, but under the surface, that's what's doing the work for us, as both `prefix` and `suffix` are defined using `append`.

2. Reversing Lists

Append is a useful predicate but it can be a source of inefficiency, and that you don't want to use it all the time.
The source of its inefficiency is: `append` doesn't join two lists in one simple action. Rather, it needs to work its way down its first argument until it finds the end of the list, and only then can it carry out the concatenation.

If we have two lists and we just want to concatenate them, this is probably not too problematic. But matters may be very different if the first two arguments are given as *variables*. As we've just seen, it can be very useful to give `append` variables in its first two arguments, for this lets Prolog search for ways of splitting up the lists. But there is a price to pay: a lot of search is going on, and this can lead to very inefficient programs.

Let us now define `reverse` predicate using two different methods.

2.1. Naive Reverse using append

Here's a recursive definition of what is involved in reversing a list:
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 with `[H]`.

With the help of `append` it is easy to turn this recursive definition into Prolog:

```
naiverev([],[]).
naiverev([H|T],R) :- naiverev(T,RevT),append(RevT,[H],R).
```

Try to `trace` a query for this predicate. The trace is very instructive with the program spending a lot of calling `append`. The result is thus, very inefficient. There is a better way of reversing lists.

2.2 Reverse using accumulator

The better way is to use an accumulator. The accumulator will be a list, and when we start it will be empty.

Suppose we want to reverse `[a,b,c,d]`. At the start, our accumulator will be `[]`. So we simply take the head of the list we are trying to reverse and add it as the head of the accumulator. We then carry on processing the tail, thus we are faced with the task of reversing `[b,c,d]`, and the accumulator is `[a]`. Again we take the head of the list we are trying to reverse and add it as the head of the accumulator (thus the new accumulator is `[b,a]`) and carry on trying to reverse `[c,d]`. Again we use the same idea, so we get a new accumulator `[c,b,a]`, and try to reverse `[d]`. The next step yields an accumulator `[d,c,b,a]` and the new goal of trying to reverse `[]`. This is where the process stops: and *the accumulator contains the reversed list we want*.

Here's the accumulator code:

```
accRev([H|T],A,R) :- accRev(T,[H|A],R).
accRev([],A,A).
```

The recursive clause is responsible for chopping of the head of the input list, and pushing it onto the accumulator. The base case halts the program, and copies the accumulator to the final argument.
It's a good idea to write a predicate which carries out the required initialization of the accumulator for us:

```
rev(L,R) :- accRev(L,[],R).
```

Try to trace this predicate after posing a query. It is clearly better than `naiverev` as it is much less instructive.

3. Exercises

1. Let's call a list doubled if it is made of two consecutive blocks of elements that are exactly the same. For example, `[a,b,c,a,b,c]` is doubled (it's made up of `[a,b,c]` followed by `[a,b,c]`) and so is `[foo,gubble,foo,gubble]`. On the other hand, `[foo,gubble,foo]` is not doubled.

Write a predicate `doubled(List)` which succeeds when `List` is a doubled list.

2. Write a predicate `palindrome(List)`, which checks whether `List` is a palindrome. For example, to the queries
   `?- palindrome([r,o,t,a,t,o,r]).`
   and
   `?- palindrome([n,u,r,s,e,s,r,u,n]).`
   Prolog should respond 'yes', but to the query
   `?- palindrome([n,o,t,h,i,s]).`
   Prolog should respond 'no'.

3. Write a predicate `second(X,List)` which checks whether `X` is the second element of `List`.

4. Write a predicate `swap12(List1,List2)` which checks whether `List1` is identical to `List2`, except that the first two elements are exchanged.

5. Write a predicate `final(X,List)` which checks whether `X` is the last element of `List`.

6. Write a one line definition of the `member` predicate by making use of `append`. How does this new version of member compare in efficiency with the standard one?

7. Write a predicate `toptail(InList,Outlist)` which says 'no' if `Inlist` is a list containing fewer than 2 elements, and which deletes the first and the last elements of `Inlist` and returns the result as `Outlist`. For example:
   ```
   toptail([a],T).
   no
   toptail([a,b],T).
   T=[]
   toptail([a,b,c],T).
   T=[b]
   ```

Hint: here's where append comes in useful.

8. Write a predicate `swapfl(List1,List2)` which checks whether `List1` is identical to `List2`, except that the first and last elements are exchanged.
   Hint: here's where append comes in useful again.

9. Here is an exercise on logic puzzles:
   There is a street with three neighboring houses that all have a different color. They are red, blue, and green. People of different nationalities live in different houses and they all have a different pet.
   Here are some more facts about them:
   - *The Englishman lives in the red house.*
   - *The jaguar is the pet of the Spanish family.*
   - *The Japanese person lives to the right of the snail keeper.*
   - *The snail keeper lives to the left of the blue house.*

   Who keeps the zebra?
   Define a predicate `zebra/1` that tells you the nationality of the owner of the zebra.
   Hint: Think of a representation for the houses and the street. Code the four constraints in Prolog. `member` *and* `sublist` *might be useful predicates.*

10. Write a predicate `set(InList,OutList)` which takes as input an arbitrary list, and returns a list in which each element of the input list appears only once. For example, the query
    `set([2,2,foo,1,foo, [],[]],X).`
    should yield the result
    `X = [2,foo,1,[]].`
    Hint: use the member predicate to test for repetitions of items you have already found.

11. We 'flatten' a list by removing all the square brackets around any lists it contains as elements, and around any lists that its elements contain as element, and so on for all nested lists. For example, when we flatten the list `[a,b,[c,d],[[1,2]],foo]` we get the list

`[a,b,c,d,1,2,foo]` and when we flatten the list
`[a,b,[[[[[[[c,d]]]]]]],[[1,2]],foo,[]]`
we also get `[a,b,c,d,1,2,foo].`
Write a predicate `flatten(List,Flat)` that holds when the first argument List flattens to the second argument `Flat`. This exercise can be done without making use of `append`.