# Prolog Lists

David Woods
dwoods@scss.tcd.ie

Week 8 - HT

## What is a List?

A list in Prolog is a collection of terms, which is useful for grouping items together, or for dealing with large volumes of related data, etc.

### Examples

1. `[red, white, black, yellow]`

   Lists are enclosed by square brackets, and items are separated by commas. The length of a list is the number of items it contains. The length of this list is **4**.

2. `[nick, nefarious(nancy), N, 10, nick]`

   Lists can contain repeated items, and items can be any kind of Prolog term, including: atoms, numbers, variables, and complex terms. The length of this list is **5**.

3. `[]`

   This is a special list known as the **empty list**. It has no items, no internal structure, and is length **0**.

4. `[nick, [michael, mary], [lucy, lucky(liam)]]`

   Lists can contain other lists as items. The length of this list is **3**, as each sublist is counted as one item.

5. `[[], alive(adam), [2, [tom, tim]], [], Var, [2, [tom, tim]]]`

   This list combines the features from the previous examples. Its length is **6**.

## Extracting Information

We can think of a list as being made up of two parts: the first element, known as the **Head**, and everything else, called the **Tail**. Prolog uses a built-in operator, the **pipe** ( `|` ) in order to give us this split for a list.
If we use unification with the first example list above and this new operator, as follows:

```
[Head|Tail] = [red, white, black, yellow].
```

we will get this output:

```
Head = red
Tail = [white, black, yellow]
```

It is important to notice that the Tail of the list is itself also a list, and therefore, we can repeat this procedure as long as there are elements left in the Tail.

We can also specify that we want to extract more than one item from the beginning of the list, by using multiple variables before the pipe. Everything that remains will become the Tail.

```
[H1, H2|T] = [red, white, black, yellow].
```

```
H1 = red
H2 = white
T = [black, yellow]
```

We can use the anonymous variable here if we want to just extract particular items. For example, `[_, H2, _, H4|_]` will extract just the second and fourth items of a list, discarding the rest.

One very important fact is that trying to unify `[H|T]` with `[]` will **fail**. This is because the empty list has *no internal structure*, and cannot be split into a Head and Tail. This is extremely important when you want to recurse through a list.

# Recursion with Lists

The fact that the Tail of a list is also a list makes it easy to see how you can recurse through all of the elements of a list. Since the empty list cannot be split into a Head and Tail, it will often serve as a way of stopping the recursion going on forever, like a base case should.

## Example 1

We're working as a bouncer at a nightclub, and we want to write a predicate which will let us know whether someone should be allowed in or not, based on whether their name is on the guestlist. If their name is on the list, then they should be permitted to enter. Otherwise, they will have to wait outside.

Since we need to recurse through the guestlist, we will first need a base case. For this example, we can say that if the person's name is the first one on the list, then they should be let in straight away without having to check any other names:

```
onTheGuestlist(Name, [Name|RestOfTheList]).
```

If the person was not the first name on the guestlist, we need to start checking the rest of it. We should write our recursive case in order to look at the other names:

```
onTheGuestList(Name, [FirstPerson|Rest]) :- onTheGuestList(Name, Rest).
```

And that's all we need to do. If the person is the first name on the list, they will be allowed in. Otherwise, we discard the first name, and check the second name, which acts like the new first name in our shortened list. If that one's no good, we keep going until the list is empty. If we get to that point, we can no longer split the list into a Head and a Tail, so we know the person's name was not on the guestlist, and they should not be allowed in.

## Example 2

We want to write a predicate which will ensure that there is always a balance between yin and yang. If there is too much of one or the other, then the world will be thrown into chaos, and we don't want that. We should also ensure that we have *only* yin and yang in our balance, and not their cousin, yon.

Our base case will handle the situation where there is no yin, and no yang:

```
yinAndYang([], []).
```

In this scenario, we have balance: 0 on the left, and 0 on the right. Now we have to deal with the more complex situation where we have both yin and yang in our lists:

```
yinAndYang([yin|Tail1], [yang|Tail2]) :- yinAndYang(Tail1, Tail2).
```

Once again, that's all we need to do in order to maintain balance in the world. This recursive predicate will take the first yin off the left-hand list, and the first yang off the right-hand list, and then compare the two list-tails to make sure that they are also balanced.

This way, we don't have to worry about yon getting into the mix, since he won't be able to unify with either of the list-heads in the rule-head of the recursive rule.

We can also use this predicate to generate the appropriate amount of yin or yang in order to create balance:

```
?- yinAndYang([yin, yin, yin], Y).
```

will return **`Y = [yang, yang, yang]`**.

## Re-useable Lists

Note that if you want to store a list in your knowledge base, you can do so by writing a complex term with your list as an argument into your Prolog file. For example:

```
peopleList([alice, bob, carol, diane, eddy]).
```

You can then use it at the Prolog prompt by using a variable to extract the list:

```
?- peopleList(List), enumerate(List, X).
```

Do not be tempted to write something into your knowledge base like `List = [a, b, c]`, as this is an *evaluation*, and **not** an assignment. It will not store the list in the variable as you might expect. Use the complex term method to achieve this effect.

# Lab Exercise

You are required to write three predicates (outlined below) which will deal with lists in a Prolog file. These predicates should be defined recursively in order to function correctly. Make sure your base cases are well-defined!

You should email your solutions to me by email (**dwoods@scss.tcd.ie**) by 23:59 on Monday 13th March, 2017.

*Be sure to add comments to your code explaining what it does or you will **not** receive full marks!*

1. **`member/2`**

   This predicate will take two arguments, where the second should be a list. The predicate should return **`true`** if the first argument exists as an item of the list.

   Examples:

   (a) `member(1, [4, 9, 8, 4, 1, 3]).`
   Should return **`true`**.

   (b) `member(pluto, [mercury, venus, earth, mars, jupiter, saturn, uranus, neptune]).`
   Should return **`false`**.

   (c) `member(Var, [1, two, Three, item(4), [five]]).`
   Should return each element of the list, in order, unified with **`Var`**.

2. **`enumerate/2`**

   This predicate should take a list as the first argument, and return an integer number as the second argument, representing the length of the list, or the number of items it contains.

   Examples:

   (a) `enumerate([john], R).`
   Should return **`R = 1`**.

   (b) `enumerate([], R).`
   Should return **`R = 0`**.

   (c) `enumerate([john, alice, Maisy, 2017], R).`
   Should return **`R = 4`**.

3. **`combine/3`**

   This predicate should take two lists of nodes as the first two arguments, and should return a list of the combined elements with matching indices for the third argument, using `jump/2` to group the paired nodes. If a pair cannot be found for a node, then the combination is invalid.

   Examples:

   (a) `combine([node1, node2], [node3, node4], Combined).`
   Should return **`Combined = [jump(node1, node3), jump(node2, node4)]`**.

   (b) `combine([node1, node2], [node3], Combined).`
   Should return **`false`**.

   (c) `combine([node1, node2], [node3, node4, node5], Combined).`
   Should return **`false`**.