Tutorial Sheet - 4

Lists

1. Lists

A list is a finite sequence of elements. Here are some examples of lists in Prolog:

- `[mia, vincent, jules, yolanda]`
- `[mia, robber(honey_bunny), X, 2, mia]`
- `[]`
- `[mia, [vincent, jules], [butch, girlfriend(butch)]]`
- `[[], dead(zed), [2, [b, chopper]], [], Z, [2, [b, chopper]]]`

We can learn some important things from these examples.

1. We can specify lists in Prolog by enclosing the elements of the list in square brackets (that is, the symbols `[` and `]`). The elements are separated by commas. For example, our first example `[mia, vincent, jules, yolanda]` is a list with four elements.The length of a list is the number of elements it has, so our first example is a list of length four.
2. From our second example, `[mia,robber(honey_bunny),X,2,mia]`, we learn that all sorts of Prolog objects can be elements of a list. The first element of this list is `mia`, an atom; the second element is `robber(honey_bunny)`, a complex term; the third element is `X`, a variable; the fourth element is `2`, a number. Moreover, we also learn that the same item may occur more than once in the same list: for example, the fifth element of this list is `mia`, which is the same as the first element.
3. The third example shows that there is a very special list, the **empty list**. The empty list (as its name suggests) is the list that contains no elements. The length of the empty list is Zero (for the length of a list is the number of members it contains, and the empty list contains nothing).
4. The fourth example teaches us something extremely important: lists can contain other lists as elements. For example, the second element of `[mia, [vincent, jules], [butch,girlfriend(butch)]]`

is the list `[vincent,jules]`, and the third element is `[butch,girlfriend(butch)]`. In short, lists are examples of recursive data structures: lists can be made out of lists. The length of the fourth list is three. The elements of the list are the things between the outermost square brackets separated by commas. So this list contains three elements.

5. The last example mixes all these ideas together. We have here a list which contains the empty list (in fact, it contains it twice), the complex term `dead(zed)`, two copies of the list `[2, [b, chopper]]`, and the variable `Z`. Note that the third (and the last) elements are lists which themselves contain lists (namely `[b, chopper]`).

Any non-empty list can be thought of as consisting of two parts: the **head** and the **tail**. The head is simply the first item in the list; the tail is everything else. Or more precisely, the tail is the list that remains when we take the first element away, i.e. the tail of a list is always a list again.

For example, the head of `[mia, vincent, jules, yolanda]` is `mia` and the tail is `[vincent, jules, yolanda]`.

Similarly, the head of `[[], dead(zed), [2, [b, chopper]], [], Z, [2, [b, chopper]]]` is `[]`, and the tail is `[dead(zed), [2,[b,chopper]],[],Z,[2,[b, chopper]]]`.

For the list `[dead(zed)]`, the head is the first element of the list, which is `dead(zed)`, and the tail is the list that remains if we take the head away, which, in this case, is the empty list `[]`.

Note that only non-empty lists have heads and tails. That is, the empty list contains no internal structure. For Prolog, the empty list [] is a special, particularly simple, list.

Prolog has a special inbuilt operator `|` which can be used to **decompose a list into its head and tail**. It is a key tool for writing Prolog list manipulation programs.

The most obvious use of | is to extract information from lists. We do this by using `|` together with matching. For example, to get hold of the head and tail of `[mia,vincent, jules,yolanda]` we can pose the following query:

```
?- [Head| Tail] = [mia, vincent, jules, yolanda].
Head = mia
Tail = [vincent,jules,yolanda]
yes
```

That is, the head of the list has become bound to `Head` and the tail of the list has become bound to `Tail`.

Note that there is nothing special about `Head` and `Tail`, they are simply variables. We could just as well have posed the query:

```
?- [X|Y] = [mia, vincent, jules, yolanda].
X = mia
Y = [vincent,jules,yolanda]
yes
```

As mentioned above, only non-empty lists have heads and tails. If we try to use `|` to pull `[]` apart, Prolog will fail:

```
?- [X|Y] = [].
no
```

That is, Prolog treats `[]` as a special list.

Let's look at some other examples:

```
?- [X|Y] = [[], dead(zed), [2, [b, chopper]], [], Z].
X = []
Y = [dead(zed),[2,[b,chopper]],[],_7800]
Z = _7800
yes
```

But we can can do a lot with `|`; it really is a very flexible tool. For example, suppose we wanted to know what the first two elements of the list were, and also the remainder of the list after the second element. Then we'd pose the following query:

```
?- [X,Y | W] = [[], dead(zed), [2, [b, chopper]], [], Z].
X = []
```

```
Y = dead(zed)
W = [[2,[b,chopper]],[],_8327]
Z = _8327
yes
```

Another useful operator is _. It is also known as an **anonymous variable**. Suppose we were interested in getting hold of the second and fourth elements of the list: `[[], dead(zed), [2, [b, chopper]], [], Z]`.
Now, we could find out like this:

```
?- [X1,X2,X3,X4 | Tail] = [[], dead(zed), [2, [b, chopper]],
[], Z].
X1 = []
X2 = dead(zed)
X3 = [2,[b,chopper]]
X4 = []
Tail = [_8910]
Z = _8910
yes
```

However, we've got a lot of extra information too (namely the values bound to `X1`, `X3` and `Tail`). If we're not interested in this information, there is a simpler way to obtain only the information we want: we can pose the following query instead:

```
?- [_,X,_,Y|_] = [[], dead(zed), [2, [b, chopper]], [], Z].
X = dead(zed)
Y = []
Z = _9593
yes
```

The _ symbol (that is, *underscore*) is the anonymous variable. We use it when we need to use a variable, but we're not interested in what Prolog instantiates it to. As seen in the above example, Prolog does not bother telling us what _ was bound to. Moreover, note that *each occurrence of _ is independent: each is bound to something different*. This cannot happen with an ordinary variable. It's simply a way of telling Prolog to bind something to a given position, completely independently of any other bindings.

Let's look at one last example. The third element of our working example is a list. Suppose we wanted to extract the tail of this internal list, and that we are not interested in any other information.The query would be as follows:

```
?- [_,_,[_|X]|_] =
               [[], dead(zed), [2, [b, chopper]], [], Z, [2,
               [b, chopper]]].
X = [[b,chopper]]
Z = _10087
yes
```

2. <u>Member</u>

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 program that, when given as inputs an arbitrary object `X` and a list `L`, tells us whether or not `X` belongs to `L`.

The program that does this is usually called member, and it is the simplest example of a Prolog program that exploits the recursive structure of lists. Here it is:

```
member(X,[X|T]).
member(X,[H|T]) :- member(X,T).
```

Let's take a closer look.
We'll start by reading the program declaratively. And read this way, it is obviously sensible. The first clause (the fact) simply says: an object `X` is a member of a list if it is the head of that list. Note that we used the inbuilt `|` operator to state this (simple but important) principle about lists.
The recursive rule says: an object `X` is a member of a list if it is a member of the tail of the list. Again, note that we used the `|` operator to state this principle.

Suppose we posed the following query:

```
?- member(yolanda,[yolanda,trudy,vincent,jules]).
```

Prolog will immediately answer 'Yes'. Why? Because it can unify `yolanda` with both occurrences of `X` in the first clause (the fact) in the definition of `member/2`, so it succeeds immediately.

Now consider the following query:
`?- member(vincent,[yolanda,trudy,vincent,jules]).`
Now the first rule fails so Prolog goes to the second clause, the recursive rule. This gives Prolog a new goal: it now has to see if
`member(vincent,[trudy,vincent,jules]).`
Again the first clause won't help, so Prolog goes to the recursive rule. This gives it a new goal, `member(vincent,[vincent,jules]).`
This time, the first clause does help, and the query succeeds.

What about when `X` is not a member of the list? Use `trace` mode to find out.

To summarize the `member/2` it is a recursive predicate, which systematically searches down the length of the list for the required item. It does this by stepwise breaking down the list into smaller lists, and looking at the first item of each smaller list. The reason that this recursion is safe (that is, the reason it does not go on forever) is that at the end of the line Prolog has to ask a question about the empty list. The empty list cannot be broken down into smaller parts, and this allows a way out of the recursion.

It can also be used to find out the members of a list:
```
member(X,[yolanda,trudy,vincent,jules]).
X = yolanda ;
X = trudy ;
X = vincent ;
X = jules ;
no
```

3. Recursing down lists

Let's suppose we need a predicate `a2b/2` that 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. For example, if we pose the following query
`a2b([a,a,a,a],[b,b,b,b]).`
we want Prolog to say 'yes'. On the other hand, if we pose the
`a2b([a,a,a,a],[b,b,b]).`
or the query

```
a2b([a,c,a,a],[b,b,5,4]).
```
we want Prolog to say ‘no’.

We give the definition:

```
a2b([],[]).
a2b([a|Ta],[b|Tb]) :- a2b(Ta,Tb).
```
The first clause records the fact that the empty list contains exactly as many `a`s as `b`s. The second clause says: the `a2b/2` predicate should succeed if its first argument is a list with head `a`, its second argument is a list with head `b`, and `a2b/2` succeeds on the two tails.

Try posing the following queries on the predicate and understand their working using `trace` mode.

```
a2b([a,a,a],[b,b,b]).
a2b([a,a,a,a],[b,b,b]).
a2b([a,c,a,a],[b,b,5,4]).
a2b([a,a,a,a],X).
a2b(X,Y).
```

Note that a predicate has been successfully implemented if it is capable of finding out not only if the query is true but also when it is false and is able to give answers for sufficiently instantiated queries.

4. Exercises

1. How does Prolog respond to the following queries?
    a. [a,b,c,d] = [a,[b,c,d]].
    b. [a,b,c,d] = [a|[b,c,d]].
    c. [a,b,c,d] = [a,b,[c,d]].
    d. [a,b,c,d] = [a,b|[c,d]].
    e. [a,b,c,d] = [a,b,c,[d]].
    f. [a,b,c,d] = [a,b,c|[d]].
    g. [a,b,c,d] = [a,b,c,d,[]].
    h. [a,b,c,d] = [a,b,c,d|[]].
    i. [] = _.
    j. [] = [_].

k. [] = [_|[]].

2. Suppose we are given a knowledge base with the following facts:
```
tran(eins,one).
tran(zwei,two).
tran(drei,three).
tran(vier,four).
tran(fuenf,five).
tran(sechs,six).
tran(sieben,seven).
tran(acht,eight).
tran(neun,nine).
```
   Write a predicate `listtran(G,E)` which translates a list of German number words to the corresponding list of English number words. For example:
```
listtran([eins,neun,zwei],X).
```
   should give:
```
X = [one,nine,two].
```
   Your program should also work in the other direction. For example, if you give it the query
```
listtran(X,[one,seven,six,two]).
```
   it should return:
```
X = [eins,sieben,sechs,zwei].
```
   Hint: to answer this question, first ask yourself 'How do I translate the empty list of number words?'. That's the base case. For non-empty lists, first translate the head of the list, then use recursion to translate the tail.

3. Write a predicate `twice(In,Out)` whose left argument is a list, and whose right argument is a list consisting of every element in the left list written twice. For example, the query
```
twice([a,4,buggle],X).
```
   should return
```
X = [a,a,4,4,buggle,buggle]).
```
   And the query
```
twice([1,2,1,1],X).
```
   should return

```
X = [1,1,2,2,1,1,1,1]
```

4. Write a 3-place predicate `combine1` which takes three lists as arguments and combines the elements of the first two lists into the third as follows:
```
?- combine1([a,b,c],[1,2,3],X).
X = [a,1,b,2,c,3]
?- combine1([foo,bar,yip,yup],
[glub,glab,glib,glob],Result).
Result = [foo,glub,bar,glab,yip,glib,yup,glob
```

5. Now write a 3-place predicate `combine2` which takes three lists as arguments and combines the elements of the first two lists into the third as follows:
```
?- combine2([a,b,c],[1,2,3],X).
X = [[a,1],[b,2],[c,3]]
?- combine2([foo,bar,yip,yup],
[glub,glab,glib,glob],Result).
Result = [[foo,glub],[bar,glab],[yip,glib],[yup,glob]]
```

6. Finally, write a 3-place predicate `combine3` which takes three lists as arguments and combines the elements of the first two lists into the third as follows:
```
?- combine3([a,b,c],[1,2,3],X).
X = [join(a,1),join(b,2),join(c,3)]
?- combine3([foo,bar,yip,yup],
[glub,glab,glib,glob],R).
R = [join(foo,glub),join(bar,glab),
join(yip,glib),join(yup,glob)]
```

7. Write a predicate `mysubset/2` that takes two lists (of constants) as arguments and checks, whether the first list is a subset of the second.

8. Write a predicate mysuperset/2 that takes two lists as arguments and checks, whether the first list is a superset of the second.