Worksheet 5

Lists

Now that we have covered simple Prolog programming, we need to start doing interesting things with it. Introducing some useful data structures would seem like a good first step.

The aims of this worksheet are

- that you see that lists in logic programming languages do not require any new constructs to be added to the language, but are just an exploitation of the function symbols;
- that you wrap your head further around the sorts of Prolog programs that are used to manipulate recursive data structures.

5.1 Representing Lists

We can do some really clever things with the function symbols we introduced in worksheet 4. We can build up quite complicated data structures. For example, we can build up lists of data:

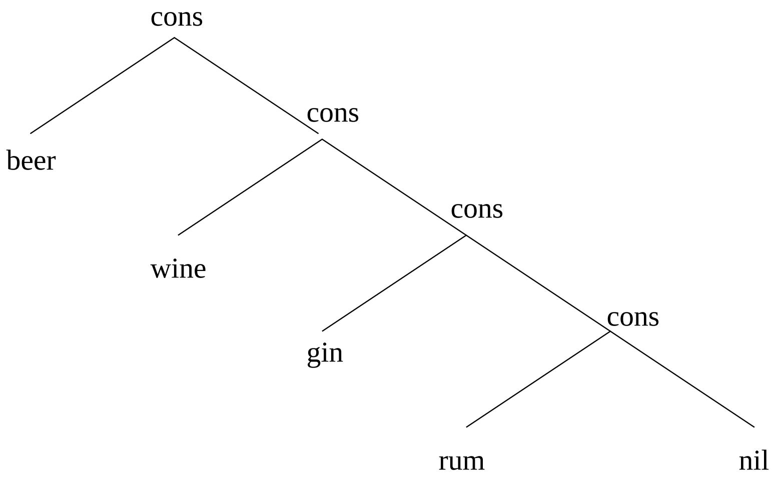
```
likes(alf, cons(beer, cons(wine, cons(gin, cons(rum, nil))))).
```

Here, `likes` is a predicate symbol whose arity is 2, being used to relate a person with a list of the drinks s/he likes. So, in this particular fact the person is `alf`, and the list of drinks is represented by a complex term, `cons(beer, cons(wine, cons(gin, cons(rum, nil))))`. Let's look at how this complex term is constructed.

`cons` is a function symbol which we've *invented* (i.e. I made it up; it is not built into Prolog) which, we've decided, takes two terms as its arguments. In the above example, the first argument is `beer` and the second argument is another complicated term: `cons(wine, cons(gin, cons(rum, nil)))`. This complex term is itself built from an application of the `cons` function to two arguments: the constant `wine` and yet another complicated term: `cons(gin, cons(rum, nil))`. And this complex term is itself built from an application of `cons` to two arguments: `gin` and the complex term `cons(rum, nil)`. This final complex term is an application of the function `cons` to two arguments (both constants this time): `rum` and `nil`. `nil` is a symbol we've invented (i.e. I made it up again; it is not built into Prolog) to indicate the empty list.

You can see that a list of objects can be captured using a two-place function symbol `cons` and a constant `nil`, and the simple idea of nesting uses of the function symbol within other uses of the function symbol.

The complex term can be shown pictorially as follows:



In fact, we didn't need to invent the symbols `cons` and `nil` because Prolog has an in-built function and an in-built constant. The function symbol that Prolog uses is not `cons` but is simply a full-stop (`.`) and the constant for the empty list is not `nil` but is simply a pair of square brackets (`[]`). So, while there was nothing wrong with the original formulation of our list, we can use Prolog's own symbols instead:

```
likes(alf, .(beer, .(wine, .(gin, .(rum, []))))).
```

This is shorter than before but both are ugly. Therefore, Prolog has provided some syntactic sugar to make it easier to read:

```
likes(alf, [beer, wine, gin, rum]).
```

Here are some equivalences between the ugly notation and the syntactic sugar:

[]	[]
.(a, [])	[a]
.(a, .(b, []))	[a, b]
.(a, .(b, .(c, [])))	[a, b, c]

Suppose you've represented some `likes` facts in the Prolog database:

```
likes(alf, [beer, wine, gin, rum]).
likes(deb, []).
likes(fay, [scotch]).
likes(ian, [coke, water, tea, lager]).
likes(pat, [wine, beer, tea]).
```

We can now query these facts. For example, the following queries ask for people who like no drinks, exactly one drink, exactly two drinks, exactly three drinks, and exactly seven drinks respectively:

```
?- likes(P, []).
P = deb ;
no

?- likes(P, [D1]).
D1 = scotch
P = fay ;
no

?- likes(P, [D1, D2]).
no

?- likes(P, [D1, D2, D3]).
D3 = tea
D2 = beer
D1 = wine
P = pat ;
no

?- likes(P, [D1, D2, D3, D4, D5, D6, D7]).
no
```

The following gives people who like any number of drinks (zero or more):

```
?- likes(P, Ds).
Ds = [beer, wine, gin, rum]
P = alf ;
```

```
Ds = []
L = deb
```

etc.

Let's set up the correspondences with the original notation:

English	Sugar	Original
any number of drinks	`likes(P, Ds)`	`likes(P, Ds)`
no drinks	`likes(P, [])`	`likes(P, [])`
exactly one drink	`likes(P, [D1])`	`likes(P, .(D1, []))`
exactly two drinks	`likes(P, [D1, D2])`	`likes(P, .(D1, .(D2, [])))`
exactly three drinks	`likes(P, [D1, D2, D3])`	`likes(P, .(D1, .(D2, .(D3, []))))`

So we can ask for people who like any number of drinks, exactly one drink, exactly two, etc. What we can't yet do is to ask for those who like *at least one* drink (i.e. one or more).

In the original notation, when we asked for exactly one drink, we used `.(D1, [])`, i.e. a list consisting of an item and an empty list. To get one or more items, we want an item and then an arbitrary list (something that is possibly the empty list, possibly a list of one item, possibly a list of two items, etc.). So the following term is what is needed:

```
.(D1, Ds)
```

`D1` is the one item and `Ds` is an arbitrary list (could be `[]`, could be one item, two items, etc.).

```
?- likes(L, .(D1, Ds)).
Ds = [wine, gin, rum]
D1 = beer
P = alf ;

Ds = []
D1 = scotch
L = fay
```

etc.

If we want to get people who like *two or more* drinks the term would be

```
.(D1, .(D2, Ds))
```

as in the following:

```
?- likes(P, .(D1, .(D2, Ds))).
Ds = [gin, rum]
D2 = wine
D1 = beer
P = alf
```

etc.

Clearly we would like some syntactic sugar for these. But the sugar cannot convert `.(D1, Ds)` into `[D1, Ds]` and `.(D1, .(D2, Ds))` into `[D1, D2, Ds]` because `[D1, Ds]` already means exactly two items and `[D1, D2, Ds]` already means exactly three items.

We introduce a new symbol into the syntactic sugar, the vertical bar |. The bar is used to separate an item from a *list* (whereas the comma separates an item from an *item*). So the correspondences are:

English	Sugar	Original
at least one drink	`likes(P, [D1 | Ds])`	`likes(P, .(D1, Ds))`
at least two drinks	`likes(P, [D1, D2 | Ds])`	`likes(P, .(D1, .(D2, Ds))`
at least three drinks	`likes(P, [D1, D2, D3 | Ds])`	`likes(P, .(D1, .(D2, .(D3, Ds))))`

This notation regularly confuses students. The crucial thing to remember is that what follows the bar will be a list (or a variable that can be instantiated to a list). The following are therefore illegal:

```
[D1 | D2, D3]
[D1, D2 | D3, D4]
```

Their use would result in a syntax error. If you really want to write something like these (and chances are that you don't!) then you must make the arguments after the bars into lists, e.g.

`[D1	[D2, D3]]`	(This matches a list of exactly three items, so the simpler notation `[D1, D2, D3]` could be used instead.)
`[D1, D2	[D3, D4]]`	(This matches a list of exactly four items, i.e. `[D1, D2, D3, D4]`; on the other hand, `[D1, D2 | [D3 | D4]]` is a list of at least three items, i.e. `[D1, D2, D3 | D4]`.)

A programmer can use both notations interchangeably, even mixing them within the same program. But when using the syntactic sugar, you should remember that this is just a notational convenience, really the list is represented as a complex term using nested function applications.

I have said that the thing following the bar must be a list. But, as we have noted before, Prolog has no real type checking. All it really checks is that there is only one object after a bar (hence it gives a syntax error if it encounters `[D1 | D2, D3]`, where there are two items after the bar, but no syntax error for `[D1 | D2]` or `[D1 | [D2, D3]]`, in both of which there is only one object after the bar, although it is a structured object in the second example). This does mean that you can get away with typing, e.g.:

```
likes(ted, [lucozade | vimto]).
```

where there is only one item after the bar, but it is not a list. A query such as `?- likes(P, [D1 | D2])` will now return that *P = ted*, *D1 = lucozade* and *D2 = vimto*, but a query such as `?- likes(P, [D1, D2])` will return *no*. Abusing Prolog's lack of type checking is all very well for simple examples such as the above, but the recursive programs we write in subsequent sections of this worksheet will all fail (return *no*) if you have abused the bar notation in this way, because they will all attempt to recurse on the tail of the list (the part after the bar) treating this tail as a list; if you've typed something that isn't a list, your program is unlikely to be able to succeed.

□ Try out all of the above in Prolog.

5.2 List Operations

Now it's easy to write some useful functions. You may need to puzzle over these for some time. Don't worry if they're not clear first time. They do require effort,

5.2.1 Checking for a list

We were just talking about type checking, so what better example to start with than one which checks whether a variable is instantiated to a list.

Our definition will reflect the structure of the data (as per one of the rules of thumb in the previous worksheet). The simplest list is the empty list, `[]`: something is a list if it simply `[]`:

```
list([]).
```

The recursive part of the definition is that something is a list is it comprises some item `X` (no matter what kind of item) stuck together with something `Xs` that is a list (which we check recursively), where the 'sticking together' is done using the full-stop function (`.`), i.e.:

```
list(.(X, Xs)) :- list(Xs).
```

More normally we use our syntactic sugar, so the definition is:

```
list([]).
list([X|Xs]) :- list(Xs).
```

This program gives results as follows:

```
?- list([]).
yes

?- list([beer]).
yes

?- list(beer).
no

?- list([beer, wine]).
yes

?- list([beer | wine]).
no
```

□ Confirm all the above, and also try it out with its argument uninstantiated and partially instantiated (try leaving the head as a variable and the tail instantiated and vice versa.) You will observe that the definition is not fully polymodal.

Of course, we ought to add calls to `list` to most of the definitions that we write in this worksheet by way of supplying some type checking. But we won't.

5.2.2 Head of a list

We will define `head`, which will be a predicate arity 2, where the first argument is a list and the second (the result, if you like) is the head of the list. (This isn't a very useful predicate.)

```
head([X|Xs], X).
```

To use this:

```
?- head([a, b, c], H).
H = a
```

Clearly there is no definition for the empty list because empty lists have no head. Calls will fail:

```
?-head([], H).
no
```

□ This definition is fully polymodal (i.e. its mode decoration is `head(?, ?)`), perhaps not too surprising since it's not recursive. Confirm this by trying out the examples from above, and also trying lots of other patterns of instantiation for both arguments.

Exercise 5.1 *Define a binary predicate, similar to* `head`*, which given a list returns its tail. This definition will consist of two clauses, one dealing with the empty list (where the tail of the empty list is taken to be the empty list) and the other dealing with the case where the list has at least one item. (Like* `head`*, this won't be recursive; check that it is fully polymodal.)*

5.2.3 Last item of a list

How would we define a predicate for returning the last item of a list? We can't predict how long the list will be. If we had the following

```
last([A], A),
last([A, B], B).
last([A, B, C], C).
etc.
```

we would not get a finite encoding of the definition.

The following (and any of its variants)

```
last([X|Y,L], L).  % Invalid Prolog
```

won't work because the thing following a bar is supposed to be a list (a single item), whereas here it is a pair of items. If you load it into Prolog, you'll get a syntax error.

The following is wrong too:

```
last([X|[Y,L]], L). % Wrong definition
```

because the term `[X | [Y, L]]` represents a list of exactly three items (as does `[X, Y, L]`), not an arbitrary length list.

What we need to do is to recurse down to the end of the list and then return it:

```
last([X], X).
last([X1, X2 | Xs], Last) :- last([X2 | Xs], Last).
```

(Note that the base case deals with a list containing one item rather than dealing with the empty list, as might be more conventional. This is because there is no last element in an empty list. This serves to emphasise that the base case is not always as straightforward as our rule of thumb from the previous worksheet would otherwise suggest.) The base case says that the last item in a list of only one item is the item itself. The recursive case, says that if the list consists of more than one item (i.e. two items `X1` and `X2` and some tail `Xs`) then you strip off the head (`X1`) and try to find the last item in what's left, i.e. you call `last` recursively on `[X2 | Xs]`.

Here's an example of its use:

```
?- last([a, b, c], L).
L = c ;
no
```

☐ Try the definition out. (Try to find the last item in a list with no items, with exactly one item, exactly two and so on. But also try some different patterns of instantiation. Not surprisingly, it will not be fully polymodal partly because we have been obliged to put the recursive call leftmost, and partly because a query such as `?- last(X, Y)` has an infinite number of answers anyway.)

5.2.4 Membership of a list

Here we define `member(Item, List)`, which will succeed if `Item` is contained in the list `List`.

The observation we need is: the item is a member of the list if either

- it is the head of the list, or
- it is a member of the tail of the list.

Thus we get a recursive definition with two clauses corresponding to the above paraphrase (note how useful the English above is when it comes to writing the definition):

```
member(X, [X|Xs]).
member(X, [Y|Ys]) :- member(X, Ys).
```

The base case shows that `X` is a member of the list if `X` is the first element of the list, i.e. if it unifies with the first element (hence we use the variable `X` twice).

The recursive case shows that `X` is a member of the list if `X` is a member of the tail of the list.

If the item is not a member at all, eventually the recursive call of `member(X, [])` will be attempted: this goal fails because it does not match with either of the above clauses.

Let's review the definitions of `list`, `last` and `member`, as they seem to show quite a few options. Most often the base clause(s) of a definition relate to either

- the empty list (as in `list`);
- a list with one element (as in `last`); or
- a list where there is something special about the head (as in `member`).

The recursive clause(s) must then deal with the remaining case where the list is not like the base case.

- If the base case was an empty list, then the recursive case is where the list isn't empty, i.e. comprises a head and a tail (`[X|Xs]`) (as in `list`);
- If the base case was a list of one item, then the recursive case is probably a list of at least two items (`[X1, X2 | Xs]`) (as in `last`), or if the base case is two items, then the recursive case will be a list of at least three items, and so on;
- If the base case is a list with something special about the head, the recursive case will probably be the case where there is nothing special about the head (as in `member`).

□ Put the definition of `member` into a file, reconsult it and try out some queries. `member` does have some polymodality. The obvious mode is, e.g., `?- member(b, [a, b, c])`, which tests for membership, but of considerable interest is, e.g., `?- member(X, [a, b, c])`, which enumerates the members of the list one by one. Of course, most other modes are not available: they launch infinite computations. Confirm that the mode is `member(?, ++)`. Actually, that's a bit simplistic because the second argument doesn't have to be fully instantiated; only the tail of the second argument has to be fully instantiated, the head can be uninstantiated.)

`member` is needed in the following exercise.

Exercise 5.2 *This exercise relates to a restaurant owned by Fritz. The clauses below can be found in the file* `zum_fritz`*, which you can copy into your own directory. (For English speakers: vorspeisen means starter, nachspeisen means dessert.)*

```
vorspeisen_available([wurstsalat, zwiebelsuppe]).
meat_available([rinderbraten, brathahnchen, schweinekotelett]).
fish_available([matjesfilet, blaue_forelle]).
nachspeisen_available([apfelstrudel, obstsalat]).

meal(H, M, D) :- hors_d_oeuvre(H), maincourse(M), dessert(D).
```

```
maincourse(M) :- meat(M).
maincourse(M) :- fish(M).

member(X, [X|Xs]).                      % Note that we must include our
member(X, [Y|Ys]) :- member(X, Ys).     % definition of member: it is
                                        % not a predefined predicate.
```

Your task is to define `hors_d_oeuvre(H)`*,* `meat(M)`*,* `fish(F)` *and* `dessert(D)`*. The clue is that, e.g.,* `H` *is an* `hors_d_oeuvre` *if it is a member of the list of* `vorspeisen_available`*.*

The following exercises will provide you with a multitude of useful little list manipulation functions, out of which you will be able to build bigger programs. The answers are amazingly obvious once you see them, but don't be too worried if it's a struggle to come up with answers for yourself.

Exercise 5.3 *Define a predicate* `add(Item, List, Newlist)`*, where* `Newlist` *consists of* `List` *with* `Item` *on the front. (This one's easy: it's not recursive.)*

Exercise 5.4 *Define a predicate* `del(Item, List, Newlist)`*, where* `Newlist` *consists of* `List` *with an occurrence of* `Item` *removed from it. (The answer has some of the flavour of* `member` *about it.)*

Exercise 5.5 *Define a predicate* `insert(Item, List, Newlist)`*, where* `Newlist` *consists of* `List` *with* `Item` *added into it somewhere (not necessarily at the front). Your definition should be given in terms of* `del` *alone (by exploiting* `del`*'s limited polymodality.)*

Exercise 5.6 *Define a predicate which checks for consecutive elements:* `next_to(X, Y, L)` *succeeds if elements* `X` *and* `Y` *are consecutive elements of list* `L` *(i.e.* `Y` *immediately follows* `X` *in* `L`*). (The answer has some of the flavour of* `last` *about it.)*

Exercise 5.7 *Define predicates* `evenlength(L)` *and* `oddlength(L)` *using mutual recursion, where* `evenlength(L)` *succeeds if* `L` *has an even number of elements in it, and* `oddlength(L)` *succeeds if* `L` *has an odd number of elements in it. (You do not need arithmetic expressions to do this.)*

Exercise 5.8 *Enter some facts into a file, such as:*

```
dict(the, le).
dict(dog, chien).
dict(chases, chasse).
dict(cat, chat).
```

Define the predicate translate, `translate(Words, Mots)`*, where* `Mots` *is a list of French words which is the (literal!) translation of the list of English words,* `Words`*.*

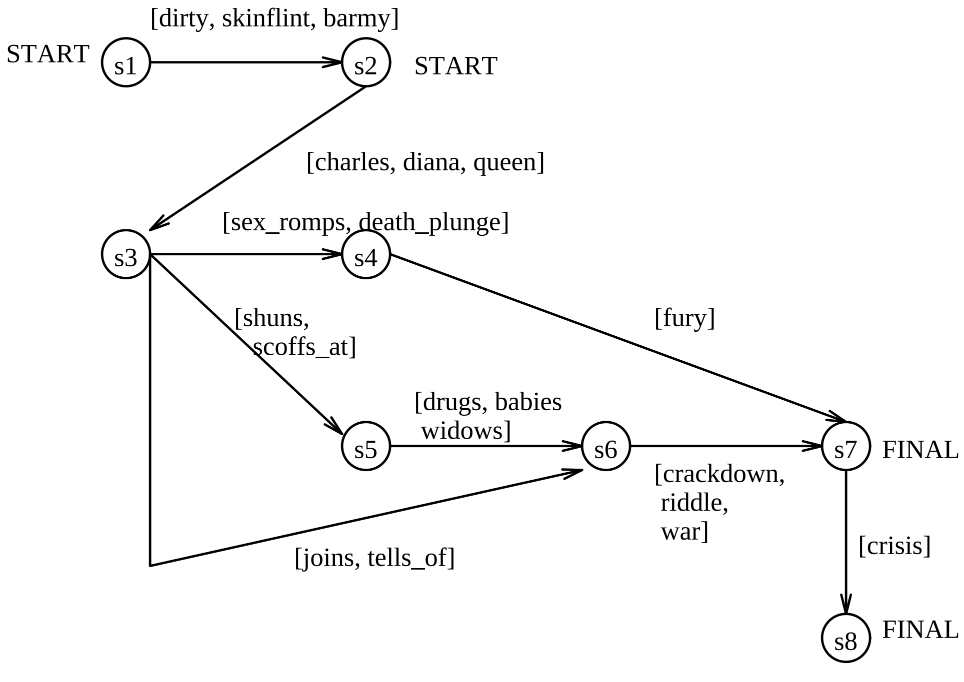
This is a pretty neat program: its polymodality means that it does English to French translation, French to English translation and also checks whether two sentences that you give it are translations of one another.

What linguistic *limitations does this machine translation program have? (Some of you might even try to overcome some of them. This part of the exercise is not something that you're obliged to do. There is no model answer. Mail any answers which you think are interesting to dgb).*

Exercise 5.9 *Get your monkey planner program from worksheet 4. This program was a bit boring because its output was only* `yes` *or* `no` *according to whether the monkey could get the banana from a given start state or not.*

Define `canget(State, Actions)` *which doesn't answer just* `yes` *or* `no` *but provides a sequence of actions that lead to success, e.g.* `Actions = [walk(at_door, at_window), push(at_window, under_banana), climb, grasp]`*.*

Exercise 5.10 *We can define a language for Sun newspaper headlines as a graph, e.g.:*



To generate headlines for tomorrow's edition of the Sun, we begin at one of the start states and forge a path through the network to one of the finals state collecting one word from each arc traversed into a list as we go. When we reach the final state, we are done (i.e. we can backtrack to find another path). For example:

```
?- headline(X).
X = [dirty, charles, sex_romps, fury] ;
X = [dirty, charles, sex_romps, fury, crisis] ;
etc.
```

The definition should be partly polymodal, so we can also use the query with its argument instantiated to see whether the input is a Sun headline or not:

```
?- headline([skinflint, queen, scoffs_at, babies, riddle]).
yes

?- headline([charles, sex_romp]).
no
```

The graph can be represented as clauses:

```
start(s1).
start(s2).
final(s7).
final(s8).
trans(s1, [dirty, skinflint, barmy], s2).
trans(s2, [charles, diana, queen], s3).
etc.
```

Define the predicate `headline(H)`*. Create some graphs of your own. (Obviously generation is likely to be infinite if you've got loops.)*

(Some of you might like to extend the whole program so that arcs can be labelled by the names of other graphs. When such an arc is traversed, you shoot off and traverse the other graph. This allows recursion in graphs. For ideas about how to do this, see G.Gazdar and C.Mellish: Natural Language Processing in Prolog*, Addison-Wesley, 1989: their chapter 2 covers the simple case in the exercise above; their chapter 3 covers the recursive case.)*

5.2.5 Appending two lists

We will define `append(L1, L2, L3)`, which will be true if `L1` and `L2` are two lists and `L3` is the result of concatenating `L2` onto the back of `L1`. The following illustrates a call:

```
?-append([a, b, c], [d, e], L).
L = [a, b, c, d, e]
```

This is obviously going to be recursive (because we need some sort of looping behaviour to keep sticking items from L2 onto the end of the result). We recurse on the first list.

The base case concerns the first list being empty. In this case, it doesn't matter what the second list is (whether it be empty, exactly one item, exactly two items, etc.). The result is simply a copy of the second list:

```
append([], Xs, Xs).
```

If the first argument is a non-empty list (i.e. it has at least one item or, in other words, it has a head of one item and a tail), i.e. `[X | Xs]`, then the result is going to have `X` as its head. Its tail is going to be the result of appending `Xs` and the second argument. (The lectures and page 72 of Bratko will make this clearer.) Hence, the recursive clause is:

```
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

(The style of this program is like that of `add` in worksheet 4.)

□ Type the two clauses in. Try different instantiations (e.g. leaving all three arguments as variables, leaving the first and third as variables, etc.). You should find that it terminates as long as the first and/or third arguments are complete lists. It doesn't terminate if the first and third arguments are variables or lists with variables as their tails, i.e. its modes are `append(++, ?, ?)` and `append(?, ?  , ++)`, although the arguments showns as `++` can have variables in their heads, but they cannot have variables in their tails. Make sure you do explore this polymodality because it is needed in the subsequent exercises of this section.

Exercise 5.11 *Define* `final` *and* `consec`*, which do the same as* `last` *and* `next_to` *respectively except that* `final` *and* `consec` *are to be defined in terms only of* `append`*. To give you an example, here is* `contains`*, which is the same as* `member` *but is defined in terms of* `append`*:*

```
contains(Item, List) :- append(X, [Item|T], List).
```

Exercise 5.12 *Define a predicate* `sublist(List1, List2)`*, which succeeds if* `List1` *is a sublist of (i.e. appears without interruption and in the right order in)* `List2`*. (Hint: Use a couple of calls to* `append`*.)*

Exercise 5.13 *Use* `append` *to define the binary predicate* `reverse`*, which holds of two arguments if one is a list that is in the reverse order of the other, e.g.:*

```
?- reverse([a, b, c], [c, b, a]).
yes
```

If you think it through, you'll probably decide that the recursive clause will contain a recursive call to `reverse` *and a call to* `append`*. (Think about it!) You'll need to think hard about which of these to put leftmost in order to get terminating computations for the most typical patterns of instantiation in queries using* `reverse`*.*

Exercise 5.14 *Give an alternative definition of* `reverse` *which does not use* `append`*. Instead, use an accumulator.*

Once you've done this, trace both definitions on a simple example and consider the difference in efficiency.

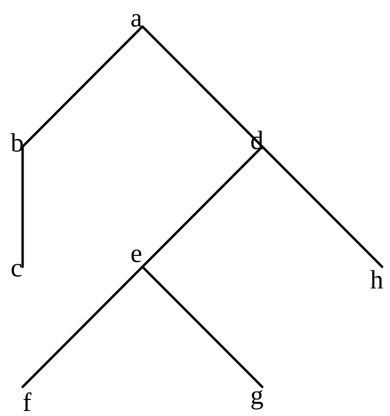
Exercise 5.15 *Define the predicate* `palindrome(L)`*, which succeeds if* `L` *is palindromic, i.e. reads the same backwards as forwards.*

5.3 Trees

So far our lists have had single items as their elements. They can of course equally well be any structures we like. For example, they could be other lists. The following nested list structure

```
[a, [b, [c]], [d, [e, [f, g]], h]]
```

effectively represents a binary tree:



Alternatively, the following term that nests the function symbol `tree` represents the same binary tree:

```
tree(a, tree(b, tree(c, void, void), void),
        tree(d, tree(e, tree(f, void, void),
              tree(g, void, void)), tree(h, void, void)))
```

Exercise 5.16 *Given the second representation of a binary tree (i.e. a nested use of* `tree`*, define* `prefix(Tree, List)`*,* `postfix(Tree, List)` *and* `infix(Tree, List)`*, which visit the nodes of the tree in pre-order, post-order and in-order respectively, accumulating the results in* `List`*. For example:*

```
?- prefix(tree(a, tree(b, void, void), tree(c, void, void)),
List).
List = [a, b, c] ;
no

?- postfix(tree(a, tree(b, void, void), tree(c, void, void)),
List).
List = [b, c, a] ;
no

?- infix(tree(a, tree(b, void, void), tree(c, void, void)),
List).
List = [b, a, c] ;
no
```

Exercise 5.17 *Write a tree membership operator for the same tree representation which determines whether the item is in the tree at any level. For example:*

```
?- tree_member(b, tree(a, tree(b, void, void), tree(c, void,
void))).
yes
```

5.4 Sorting

Now that we have lists an obvious thing to do is write some sort programs.

We will sort lists of numbers. We won't use Peano's axiomatisation of the natural numbers as that is too cumbersome. We will use some in-built, 'impure' Prolog facilities, that are far more efficient than using Peano's axiomatisation. The cost of the efficiency, as will be shown properly in a future worksheet, is a considerable loss of polymodality. (Indeed, up to now, when a definition has not been fully polymodal, what this has meant is that it launches an infinite computation for certain patterns of instantiation; any answers produced during these infinite computations have still been correct. Now, with this impure arithmetic, we actually get our first run-time errors: calling the arithmetic facilities with the wrong modes gives an error message.)

For now, just think of numbers as ordinary Prolog constants. We will need to be able to compare numbers. You can use `X > Y`, `X >= Y`, `X < Y` and `X =< Y` as goals in the body of a rule to compare numbers. Their modes are `++ < ++`, `++ >= ++`, etc., i.e. both arguments must be fully instantiated (to numbers), else you'll get an error message.

Exercise 5.18 *Here's a definition of* `perm(List1, List2)`*, which succeeds if* `List2` *is some rearrangement of* `List1`*. Obviously, on backtracking, all rearrangements are eventually found.*

```
perm([], []).
perm([X|Xs], Ps) :- perm(Xs, PTail), insert(X, PTail, Ps).
```

`insert` *is a predicate you defined in an earlier exercise. (I hope you got it right, else this won't work! Check it with me if you're worried.) (The ordering of the subgoals in the recursive clause is again a compromise between our preference not to put the recursive call leftmost versus problems with the non-full-polymodality of* `insert`*.)*

Using `perm` *gives a very simple way of writing a sort:*

```
permsort(OriginalList, SortedList) :- perm(OriginalList, SortedList),
                                      is_ordered(SortedList).
```

i.e. `SortedList` *is a sorted version of* `OriginalList` *if it is a permutation of the original list and is ordered.*

Finish the program off by defining `is_ordered(L)`*, which checks that the items of* `L` *are arranged in ascending order.*

Exercise 5.19 *Write a program that does an insertion sort. In insertion sort each element of the original list is inserted into the results list at the right position, i.e. you insert the first item of the original list into the empty list, then you insert the second item in its rightful position, then the third item, and so on.*

Exercise 5.20 *Write a program that carries out quicksorts. In quicksort, you take the head of the list that is to be sorted,* `H`*. You split the list into two sublists: the first contains items that are less than or equal to* `H` *(other than* `H` *itself); the second contains items that are greater than* `H`*. These sublists are then themselves recursively sorted. The result is given by sticking together the sorted first sublist, the head* `H` *and the sorted second sublist.*

Exercise 5.21 *Define* `merger(L1, L2, L3)`*, which takes lists that are already sorted,* `L1` *and* `L2`*, and merges them into a third sorted list* `L3`*.*

Hence, define a merge sort as follows. In quicksort the list to be sorted was split according to an arbitrary item (the head of the list). This can cause the algorithm to behave poorly. More effective is if the list is split into fairly even halves. Come up with a new definition for splitting the list that just roughly halves the list rather than using an item. Recursively sort these halves and then use `merger` *to put the results together.*

Trace all the sort programs to see their preformance. Try them on very jumbled lists and nearly sorted lists.

Exercise 5.22 *The modes of all your sorts will probably be the same, i.e.* `permsort(++, ?)`, `insertsort(++, ?)`, `quicksort(++, ?)` *and* `mergesort(++, ?)`. *None of them has the mode* `(?, ++)`. *(Confirm this.) As you confirmed this, you probably will have found that* `permsort` *came very close to having this mode. It gets all the right answers whether we use it with a goal such as* `?- permsort([3, 2, 1], X)` *or a goal such as* `?- permsort(X, [1, 2, 3])`, *the only problem being that in the latter case, after giving all the right answers, we are on an infinite computation. Try to modify* `permsort` *so that it has the mode* `permsort(?, ++)` *but retains the mode* `permsort(++, ?)`. *(Hint: replace the call to* `perm` *by a call to* `polyperm`, *i.e. a polymodal permutation program.* `polyperm` *will call the original definition of* `perm`, *but it will do something else as well.)*

Acknowledgement

Some examples and exercises are from Bratko and from Sterling & Shapiro (though most are well-known in the Prolog community). The polymodal permutation sort in the last exercise was suggested to me by Colin Runciman.