Lists

So far we have only considered simple items as arguments to our programs. However in Prolog a very common data-structure is the list.

Lists themselves have the following syntax. They always start and end with square brackets, and each of the items they contain is separated by a comma. Here is a simple list [a,freddie,A_Variable,apple]

Prolog also has a special facility to split the first part of the list (called the head) away from the rest of the list (known as the tail). We can place a special symbol | (pronounced 'bar') in the list to distinguish between the first item in the list and the remaining list. For example, consider the following.

```
[first,second,third] = [A|B]
```

where *A = first* and *B=[second,third]*

The unification here succeeds. A is bound to the first item in the list, and B to the remaining list.

List Examples 1

Here are some example simple lists

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

```
[apple,pear,bananas,breadfruit]
```

```
[ ] /*  this is a special list, it is called the empty list because  it contains nothing */
```

Now lets consider some comparisons of lists:

[a,b,c] unifies with [Head|Tail] resulting in Head=a and Tail=[b,c]

[a] unifies with [H|T] resulting in H=a and T=[]

[a,b,c] unifies with [a|T] resulting in T=[b,c]

[a,b,c] doesn't unify with [b|T]

[] doesn't unify with [H|T]

[] unifies with []. Two empty lists always match

List Example 2

Consider the following fact.

```
p([H|T], H, T).
```

Lets see what happens when we ask some simple queries.

```
?- p([a,b,c], X, Y).

X=a

Y=[b,c]

yes


?- p([a], X, Y).

X=a

Y=[]

yes


?- p([], X, Y).

no
```

Exercise 9 - Lists Exercise

Here are some problems for which unification sometimes succeeds and sometimes fails. Decide which is the case and, if you think the unification succeeds, write down the substitutions made.

1. [a,d,z,c] and [H|T] Do these unify (match) ?
2. [apple,pear,grape] and [A,pear|Rest] Do these unify (match) ?
3. [a|Rest] and [a,b,c] Do these unify (match) ?
4. [a,[]] and [A,B|Rest] Do these unify (match) ?
5. [One] and [two|[]] Do these unify (match) ?
6. [one] and [Two] Do these unify (match) ?
7. [a,b,X] and [a,b,c,d] Do these unify (match) ?

Check Your Answers

List Searching

We can use lists within facts and rules. One common way of using lists is to store information within a list and then subsequently search for this information when we run our programs. In order to search a list, Prolog

inspects the first item in a list and then goes on to repeat the same process on the rest of the list. This is done by using recursion. The search can either stop when we find a particular item at the start of the list or when we have searched the whole list, in which case the list to be searched will be the empty list. In order to do this, we have to be able to selectively pull the list apart. We have already seen how we can go about doing this. In the previous section we showed how to take the head and a tail of a list:

```
[Head|Tail]
```

This method constitutes the basis of the searching method. We shall use it to pull apart a list, looking at the first item each time, recursively looking at the tail, until we reach the empty list [], when we will stop.

List Searching: Example 1

Consider the following problem. How can I see if a particular item is on a particular list? For example I want to test to see if item apples is on the list [pears, tomatoes, apples, grapes]. One possible method of doing this is by going through the list, an item at a time, to see if we can find the item we are looking for. The way we do this in Prolog is to say that we could definitely prove an item was on a list if we knew that the target item was the first one on the list. ie.

```
on(Item,[Item|Rest]).       /* is the target item the head of the list */
```

Otherwise we could prove something was on a list if we could prove that although it didn't match the existing head of the list, it nonetheless would match another head of the list if we disregarded the first item and just considered the rest of the list i.e.

```
on(Item,[DisregardHead|Tail]):-

          on(Item,Tail).
```

We now have a program consisting of a fact and a rule for testing if something is on a rule. To recap, it sees if something is the first item in the list. If it is we succeed. If it is not, then we throw away the first item in the list and look at the rest.

List Searching: Example 2

```
on(Item,[Item|Rest]).

on(Item,[DisregardHead|Tail]):-

          on(Item,Tail).
```

Let's go through the last example again in more detail. Suppose we pose the query:

```
?- on(apples,  [pears, tomatoes, apples, grapes]).
```

The first clause of on requires the first argument of on to match with the list head. However apples and pears do not match. Thus we must move on to the second clause. This splits the list up as follows:

```
DisregardHead = pears
```

```
Tail = [tomatoes,apples,grapes]
```

This now gives us the following goal in the body of our rule:

```
on(apples, [tomatoes, apples, grapes]).
```

Again, we see if apples and tomatoes match using our initial facts. Since they don't, we again use our second clause to strip off the current list head, giving us the new goal: on(apples,[apples,grapes]). Since apples matches apples our first clause succeeds as does our query.

List Searching: Exercise

Given the following list of cities and features

```
[london_buckingham_palace,
 paris_eiffel_tower,
 york_minster,
 pisa_leaning_tower,
 athens_parthenon]
```

Write a program called member, that would be able to see if something was a member of a list. The program would inform me that sydney_opera_house was not on list by failing, yet confirm all the above were on the list by succeeding (answering yes).

List Construction.

The last example showed how we could strip off the front items on a list and recursively search the rest. We can also use the same method to build lists. For example take one existing list, say List1. We can make a new list List2 which contains List1 but with the new head prolog, as follows:

```
List2 = [prolog|List1]
```

This again is pure unification. List2 now consists of the head prolog plus whatever List1 was bound to. If List1 were to be bound e.g. List1 = [list,c,pascal,basic], List2 would now be the list [prolog,lisp,c,pascal,basic]

We can use this construction technique to build lists during recursion. We'll present three examples of this *in the forthcoming cards.*

List Construction Example 1.

An example use of list construction is when we wish to create a new list out of two existing lists. We'll illustrate this by defining a predicate called append that takes three arguments. The first two are lists. The predicate uses these two lists to produce a third list which combines the original two. For example,

```
?- append([a,b,c],[one,two,three],Result).
```

```
Result = [a,b,c,one,two,three]
```

The way we do this in Prolog uses both list construction and list deconstruction. Recall that on the previous card we saw how we could glue an item onto the front of a list to produce a new list. Thus we can produce the list [c,one,two,three] from the list [one,two three] by saying NewList = [c|[one,two,three]. This results in NewList being bound to the list [c,one,two,three]. This is the clue to how we write append in Prolog. We can recursively split the first list into single items by the deconstruction techniques discussed earlier. We can then use the construction method above to glue together a new list, which we shall illustrate next.

List Construction Example 2

In order to write append we shall first search through the first list, and taking each item in turn, add it to the second list. This we'll do recursively, by searching for the end of the first list, and then adding the items in the first list to the items in the second list in reverse order. Thus the last item in the first list is the first to be added to the second list. We illustrate this more clearly with an example. First lets see what the code to do this look like. First we must deal with the case when the first list is an empty list. In this case the result of appending the two lists will just be the second list. Thus this gives

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

Now for the rule which does most of the work. This says search through the first list can each time stick the first thing on the first list onto the resulting list.

```
append([Head|Tail],List2,[Head|Result]):-

    append(Tail,List2,Result).
```

Now lets go through in detail how this actual works.

List Construction Example 3

Recall our code and query ?- append([a,b,c],[one,two,three],Result).

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

append([Head|Tail],List2,[Head|Result]):-

    append(Tail,List2,Result).
```

Let's now follow the program as it executes. Using the second rule we first reduce the query to

```
append([b,c],[one,two,three],Result)
```

and then to

```
append([c],[one,two,three],Result)
```

and finally to

```
append([],[one,two,three],Result).
```

This final clause can match against the initial fact, giving append([],[one,two,three],[one,two,three]). Since this is a fact, this terminates the recursion. As we pop out of each recursive step we then add on (respectively) c,b, and a to the list, giving the list [a,b,c.one,two,three].

List Construction Example 4

Let's now consider a second example of constructing lists. Taking the list [1,12,3,14,5,8], lets construct a new list out of the old that contains only numbers greater than 6. To do this we can search the list as we have seen before. If the item passes our test, then we can add it to the output list. If it does not, then we can disgard the item. To make a start, if the list to search is empty, we will return the empty list. Hence we get the base case of the recursion.

```
sift([],[]).
```

Next we need to say that if the item passes our test, put it in the output list

```
sift([X|T],[X|Result]):- X > 6,    /* is X greater than 6 */
    sift(T,Result).                /* if so then go find the rest */
```

```
Otherwise we are will disgard the head and look for other hits in the tail
```

```
sift([ThrowAway|Tail],Result):-    /* disgard the head */
   sift(Tail,Result).              /* and look in the tail */
```

List Constructions Example 5

Let us now see how our program respond to the query

```
?- sift([1,12,3,14,5,8], Result).
```

To this goal, we first unify with clause 2. The result is to evaluate the subgoal 1 > 6, this clearly fails, so we move on to the third clause and try the goal sift([12,3,14,5,8],Result). This again matches on the second clause of sift. We now try the subgoal 12>6, this succeeds and we now attempt the goal sift([3,14,5,8],Result). However, notice what also happens. By using this clause we also place 12 on our output list.

Stepping further through our example, we see that greater than subgoal in clause 2 succeeds for 14 and 8, till finally we get the goal sift([],Result). This matches the first goal, with Result bound to []. As we come out of the recursion, we see that clause 2 builds up the result for to the list [8], then [14,8], then finally [12,14,8], before the query finally succeeds.

List Construction Exercises 1

Write a program to delete all reference of a particular item from a list. It should have three arguments. The list you wish to use, the item to delete, and the resulting list. Here are some example of it behaviour

```
?- delete_all([a,b,a,c,a,d],a,Result).

Result = [b,c,d]

?- delete_all([a,b,a,c,a,d],b,Result).

Result = [a,a,c,a,d]

?- delete_all([a,b,a,c,a,d],prolog,Result).

Result = [a,b,a,c,a,d]
```

List Construction Exercises 2

Write a program to replaces all occurrences of one item in a list with another. It should have four arguments. The list you wish to use. The item to replace. The item to replace it with, and the resulting list. Here are some example of its behaviour

```
?- replace_all([a,b,a,c,a,d],a,mike,Result).

Result = [mike,b,mike,c,mike,d]

?- replace_all([a,b,a,c,a,d],b,foo,Result).

Result = [a,foo,a,c,a,d]

?- replace_all([a,b,a,c,a,d],prolog,logic,Result).
```

```
Result = [a,b,a,c,a,d]
```

This is the end of the prolog tutorial.

Return to the Introduction Menu