Learn Prolog Now!

by Patrick Blackburn, Johan Bos, and Kristina Striegnitz

[next] [prev] [prev-tail] [tail] [up]

6.2 Reversing a List

The append/3 predicate is useful, and it is important to know how to put it to work. But it is just as important to know that it can be a source of inefficiency, and that you probably don't want to use it all the time.

Why is append/3 a source of inefficiency? If you think about the way it works, you'll notice a weakness: append/3 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.

Now, often this causes no problems. For example, if we have two lists and we just want to concatenate them, it's probably not too bad. Sure, Prolog will need to work down the length of the first list, but if the list is not too long, that's probably not too high a price to pay for the ease of working with append/3 .

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/3 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 searching is going on, and this can lead to very inefficient programs.

To illustrate this, we shall examine the problem of reversing a list. That is, we will examine the problem of defining a predicate which takes a list (say [a,b,c,d]) as input and returns a list containing the same elements in the reverse order (here [d,c,b,a]).

Now, a reverse predicate is a useful predicate to have around. As you will have realised by now, lists in Prolog are far easier to access from the front than from the back. For example, to pull out the head of a list L , all we have to do is perform the unification [H|_] = L ; this results in H being instantiated to the head of L . But pulling out the last element of an arbitrary list is harder: we can't do it simply using unification. On the other hand, if we had a predicate which reversed lists, we could first reverse the input list, and then pull out the head of the reversed list, as this would give us the last element of the original list. So a reverse predicate could be a useful tool. However, as we may have to reverse large lists, we would like this tool to be efficient. So we need to think about the problem carefully.

And that's what we're going to do now. We will define two reverse predicates: a naive one, defined with the help of append/3 , and a more

efficient (and indeed, more natural) one defined using accumulators.

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] .

To see that the recursive clause is correct, consider the list [a,b,c,d] . If we reverse the tail of this list we obtain [d,c,b] . Concatenating this with [a] yields [d,c,b,a] , which is the reverse of [a,b,c,d] .

With the help of append/3 it is easy to turn this recursive definition into Prolog:

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

Now, this definition is correct, but it does an awful lot of work. It is very instructive to look at a trace of this program. This shows that the program is spending a lot of time carrying out appends. This shouldn't be too surprising: after, all, we are calling append/3 recursively. The result is very inefficient (if you run a trace, you will find that it takes about 90 steps to reverse an eight element list) and hard to understand (the predicate spends most of its time in the recursive calls to append/3 , making it very hard to see what is going on).

Not nice. But as we shall now see, there is a better way.

Reverse using an accumulator

The better way is to use an accumulator. The underlying idea is simple and natural. Our 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 our 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 our 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] . Needless to say, 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 our accumulator contains the reversed list we want . To summarise: the idea is simply to work our way through the list we want to reverse, and push each element in turn onto the head of the accumulator, like this:

```
List:  [a,b,c,d]    Accumulator:  []
List:  [b,c,d]       Accumulator:  [a]
List:  [c,d]          Accumulator:  [b,a]
List:  [d]             Accumulator:  [c,b,a]
List:  []              Accumulator:  [d,c,b,a]
```

This will be efficient because we simply blast our way through the list once: we don't have to waste time carrying out concatenation or other irrelevant work.

It's also easy to put this idea in Prolog. Here's the accumulator code:

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

```
accRev([],A,A).
```

This is classic accumulator code: it follows the same pattern as the arithmetic examples we examined in the previous chapter. The recursive clause is responsible for chopping off 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.

As is usual with accumulator code, it's a good idea to write a predicate which carries out the required initialisation of the accumulator for us:

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

Again, it is instructive to run some traces on this program and compare it with naiverev/2 . The accumulator based version is clearly better. For example, it takes about 20 steps to reverse an eight element list, as opposed to 90 for the naive version. Moreover, the trace is far easier to follow. The idea underlying the accumulator based version is simpler and more natural than the recursive calls to append/3 .

Summing up, append/3 is a useful program, and you certainly should not be scared of using it. However, you also need to be aware that it is a source of inefficiency, so when you use it, ask yourself whether there is a better way. And often there is. The use of accumulators is often better, and (as the rev/2 example show) accumulators can be a natural way of handling list processing tasks.

[next] [prev] [prev-tail] [front] [up]