By now you should have a handle on programming assignment 1. However, if you are still having problems, you are probably over-thinking things. The **epoly** function can easily be written as a two-line function, and the required documentation for submission will take more space in the file than the function.

There are two cases that need to be dealt with in epoly. The first is when the list of coefficients is empty, and the question is, "What is the value of a polynomial that has no non-zero coefficients?" If you think of it in terms of writing a polynomial in either of the two standard forms, you should remember that all cases where the coefficient is zero are simply omitted. Thus, the value of an polynomial with an empty list of coefficients is zero. The second case is where there is at least one coefficient. Again, if you write a third-degree polynomial in a form like this - $f(x) = a_0 + x(a_1 + x(a_2 + xa_3))$ - you should be able to see how this can be done recursively. The recursive calls can be viewed as setting up functions as follows.

$$
\begin{aligned}
f(x) = f_0(x) &= a_0 + xf_1(x),\\
f_1(x) &= a_1 + xf_2(x),\\
f_2(x) &= a_2 + xf_3(x),\\
f_3(x) &= a_3 + xf_4(x),\\
f_4(x) &= 0.
\end{aligned}
$$

Simply view the successive functions $f_n(x)$ as nested recursive calls.

Now for some further discussion that should help with programming assignment 2.

An interesting feature of ML is the **as** keyword. This can be very useful. Consider a function (**flipcat**) that accepts two lists of **int**s, and concatenates the second to the end of a reversed copy of the first. (Don't worry about utility, as this is an example.) Here's how you use **as** in writing this function.

```
fun flipcat([], R:int list) = R
|   flipcat(L:int list as h::T, R:int list) = flipcat(T, h::R);
```

For a simple function like this, there isn't much savings in terms of typing, an little added clarity. But, not all functions in ML are simple. You can significantly improve the clarity of a long, complex function by using this feature of ML.

Notice the use of the cons operator (**::**) in the specification of the left input list as both a list, and as the head and tail of that list. If you were to write a function where you needed the first and second elements of the list individually in the body of the function, as well as the whole list, you can write that part of the input specification as **L:real list as f::s::T**. The following function uses the technique to add the first two **int**s in a list. (Note that in this case, if you type this into the ML interpreter, the operands will be assumed to be **int**s.)

```
fun addtop2([]) = []
|   addtop2([h]) = [h]
|   addtop2(h::s::T) = (h+s)::T;
```

In both functions, notice that all the potential cases of lists of **int**s are handled. While you are not required to do this for all function definitions in ML, the interpreter will complain, and you may be leaving you may be creating bugs you did not need to.

Pay attention to the cons operator's use in the results sides of the two functions. If you are concatenating a single item to the start of a list, the cons operator works. But, if you are concatenating a single item to the end of a list, it does not. Similarly, if you tried to something like **s::h::T** on the output side to exchange the positions of the first two items in the list, this would not work. The right-hand side of the cons operator must be a list, and **s::h** in the above case will not work, since **h** is not a list, and the overall operation is performed from right to left.

The following is an example where the use of the cons operator to access the first two elements of a list is used, as well as showing how to pass an operator-function to a higher-order function. This is a bubble sort, and is implemented as two functions. The recursive function **bubble** moves a single item to its correct position, while the recursive function **bsort** performs the overall sort by ensuring that each successive **bubble** places another item into a sorted list.

```
fun bubble([], Fn) = []
|   bubble([h], Fn) = [h]
|   bubble(h::s::T, Fn) =
        if Fn(h,s)
            then h::bubble(s::T, Fn)
            else s::bubble(h::T, Fn);

fun bsort([], Fn) = []
|   bsort(h::T, Fn) = bubble(h::bsort(T, Fn), Fn);

bsort([3.0,4.0,1.0, 2.0], op<);
bsort([3.0,4.0,1.0, 2.0], op>);
bsort([3.0,4.0,1.0, 2.0], op<=);
bsort([3.0,4.0,1.0, 2.0], op>=);
```

Notice that this is similar to an insertion sort in operation, but slower with more operations performed, so please do not submit this as an insertion sort. It isn't quite.