fun append([],L) = L
    | append(hd::tl,L) = hd::append(tl,L);
val append = fn : 'a list * 'a list -> 'a list

If we add the pattern
append(L,[]) = L
we get a redundant pattern warning
(Why?)

fun append ([],L) = L
    | append(hd::tl,L) = hd::append(tl,L)
    | append(L,[]) = L;

stdIn:151.1-153.20 Error: match redundant
    (nil,L) => ... 
    (hd :: tl,L) => ... 
    --> (L,nil) => ...
But a more precise decomposition is fine:

fun append ([],L) = L
| append(hd::tl,hd2::tl2) = hd::append(tl,hd2::tl2)
| append(hd::tl,[]) = hd::tl;

val append = fn :
  'a list * 'a list -> 'a list
Function Types Can be Polytypes

Recall that 'a, 'b, ... represent type variables. That is, any valid type may be substituted for them when checking type correctness.

ML said the type of append is

```ml
val append = fn :
  'a list * 'a list -> 'a list
```

Why does 'a appear in three places?

We can define eitherNull, a predicate that determines whether either of two lists is null as

```ml
fun eitherNull(L1,L2) =
  null(L1) orelse null(L2);

val eitherNull =
  fn : 'a list * 'b list -> bool
```

Why are both 'a and 'b used in eitherNull’s type?
Currying

ML chooses the most general (least-restrictive) type possible for user-defined functions.

Functions are first-class objects, as in Scheme.

The function definition
fun f x y = expression;
defines a function $f$ (of $x$) that returns a function (of $y$).

Reducing multiple argument functions to a sequence of one argument functions is called currying (after Haskell Curry, a mathematician who popularized the approach).
Thus

```haskell
fun f x y = x :: [y];
val f = fn : 'a -> 'a -> 'a list
```
says that \( f \) takes a parameter \( x \), of type \('a\), and returns a function (of \( y \), whose type is \('a\)) that returns a list of \('a\).

Contrast this with the more conventional

```haskell
fun g(x,y) = x :: [y];
val g = fn : 'a * 'a -> 'a list
```

Here \( g \) takes a pair of arguments (each of type \('a\)) and returns a value of type \('a\) list.

The advantage of currying is that we can bind one argument and leave the remaining argument(s) free.
For example

\[ f(1); \]

is a legal call. It returns a function of type

\[ \text{fn} : \text{int} \to \text{int list} \]

The function returned is equivalent to

\[
\text{fun } h \ b = 1 :: [b]; \\
\text{val } h = \text{fn} : \text{int} \to \text{int list}
\]
Map Revisited

ML supports the map function, which can be defined as

```
fun map (f, []) = []
  | map (f, x::y) =
      (f x) :: map (f, y);
```

```
val map =
  fn : ('a -> 'b) * 'a list -> 'b list
```

This type says that \texttt{map} takes a pair of arguments. One is a function from type \('a\) to type \('b\). The second argument is a list of type \('a\). The result is a list of type \('b\).

In curried form \texttt{map} is defined as

```
fun map f [] = []
  | map f (x::y) =
      (f x) :: map f y;
```

```
val map =
  fn : ('a -> 'b) ->
      'a list -> 'b list
```
This type says that $\text{map}$ takes one argument that is a function from type $'a$ to type $'b$. It returns a function that takes an argument that is a list of type $'a$ and returns a list of type $'b$.

The advantage of the curried form of $\text{map}$ is that we can now use $\text{map}$ to create “specialized” functions in which the function that is mapped is fixed.

For example,

```ml
val neg = map not;
val neg =
    fn : bool list -> bool list
neg [true,false,true];
val it = [false,true,false] : bool list
```
Let's compute power sets in ML.

We want a function \texttt{pow} that takes a list of values, viewed as a set, and which returns a list of lists. Each sublist will be one of the possible subsets of the original argument.

For example,

\[
\text{pow \ [1,2] = [[1,2],[1],[2],[[]]]}
\]

We first define a version of \texttt{cons} in curried form:

\[
\text{fun cons h t = h::t;}
\]

\[
\text{val cons = fn : 'a \to 'a list \to 'a list}
\]
Now we define $\text{pow}$. We define the powerset of the empty list, $[]$, to be $[[]]$. That is, the power set of the empty set is set that contains only the empty set.

For a non-empty list, consisting of $h::t$, we compute the power set of $t$, which we call $\text{pset}$. Then the power set for $h::t$ is just $h$ distributed through $\text{pset}$ appended to $\text{pset}$.

We distribute $h$ through $\text{pset}$ very elegantly: we just map the function $(\text{cons } h)$ to $\text{pset}$. $(\text{cons } h)$ adds $h$ to the head of any list it is given. Thus mapping $(\text{cons } h)$ to $\text{pset}$ adds $h$ to all lists in $\text{pset}$. 
The complete definition is simply

```ml
fun pow [] = [[]]
| pow (h::t) =
  let
    val pset = pow t
  in
    (map (cons h) pset) @ pset
  end;

val pow = fn : 'a list -> 'a list list
```

Let's trace the computation of

pow [1,2].

Here h = 1 and t = [2]. We need to compute pow [2].

Now h = 2 and t = [].

We know pow [] = [[]],

so pow [2] =

(map (cons 2) [[]]) @ [[]] = 
([[2]]) @ [[]] = [[2], []]
Therefore \( \text{pow} \ [1,2] = \)

\[
(\text{map} \ (\text{cons} \ 1) \ [[2],[[]]])
\]

\[
@[[2],[[]] =
\]

\[
[[1,2],[1]]@[[2],[[]] =
\]

\[
[[1,2],[1],[2],[[]]]
\]
Composing Functions

We can define a composition function that composes two functions into one:

\[
\text{fun comp } (f,g)(x) = f(g(x));
\]

\[
\text{val comp = fn :}
\]

\[
('a \to 'b) \times ('c \to 'a) \to
\]

\[
'c \to 'b
\]

In curried form we have

\[
\text{fun comp } f \ g \ x = f(g(x));
\]

\[
\text{val comp = fn :}
\]

\[
('a \to 'b) \to
\]

\[
('c \to 'a) \to 'c \to 'b
\]

For example,

\[
\text{fun sqr x:int = x\times x;}
\]

\[
\text{val sqr = fn : int } \to \text{ int}
\]

\[
\text{comp sqr sqr;}
\]

\[
\text{val it = fn : int } \to \text{ int}
\]
```sml
comp sqr sqr 3;
val it = 81 : int

In SML ◦ (lower-case O) is the infix composition operator.
Hence
sqr ◦ sqr ≡ comp sqr sqr
```