Reversing a List

- Concatenation of lists, for which we gave a recursive definition, is actually a built-in operator in SML, denoted by the symbol `@`.
- We use this operator in the following recursive definition of a function that reverses a list.

```sml
- fun reverse(L) = 
  = if L = nil then nil 
  = else reverse(tl(L)) @ [hd(L)];
val reverse = fn : 'a list -> 'a list

- reverse [1,2,3];
val it = [3,2,1] : int list
```
Definition by Patterns

- In SML functions can also be defined via patterns.
- The general form of such definitions is:

  \[
  \text{fun } \text{<identifier>}(\text{<pattern1>}) = \text{<expression1>}
  | \text{<identifier>}(\text{<pattern2>}) = \text{<expression2>}
  | \ldots
  | \text{<identifier>}(\text{<patternK>}) = \text{<expressionK>};
  \]

- where the identifiers, which name the function, are all the same, all patterns are of the same type, and all expressions are of the same type.
- Example:

  \[
  \text{- fun reverse(nil) = nil}
  = | \text{reverse(x::xs) = reverse(xs) @ [x];}
  \]

  \[
  \text{val reverse = fn : } \text{'a list -> 'a list}
  \]

Removing List Elements

- The following function removes all occurrences of its first argument from its second argument list.

  \[
  \text{- fun remove(x,L) =}
  = | \text{if (L=[])} \text{then []}
  = | \text{else if (x=hd(L)) then remove(x,tl(L))}
  = | \text{else hd(L):remove(x,tl(L));}
  = | \text{val remove = fn : } \text{"a * "a list -> "a list}
  \]

  \[
  \text{- remove(1,[5,3,1]);}
  \]

  \[
  \text{val it = [5,3] : int list}
  \]

  \[
  \text{- remove(2,[4,2,4,2,4,2,2]);}
  \]

  \[
  \text{val it = [4,4,4] : int list}
  \]
Removing List Elements

- The remove function can be used in the definition of another function that removes all duplicate occurrences of elements from its argument list:
  ```ml
  fun removedupl(L) = 
  if (L=[]) then [] 
  else hd(L)::remove(hd(L),removedupl(tl(L))); 
  val removedupl = fn : 'a list -> 'a list
  ```

Higher-Order Functions

- In functional programming languages, functions can be used in definitions of other, so-called higher-order, functions.
- The following function, apply, applies its first argument (a function) to all elements in its second argument (a list of suitable type):
  ```ml
  fun apply(f,L) = 
  if (L=[]) then [] 
  else f(hd(L))::(apply(f,tl(L))); 
  val apply = fn : ('a -> 'b) * 'a list -> 'b list
  ```
- We may apply apply with any function as argument:
  ```ml
  fun square(x) = (x:int)*x; 
  val square = fn : int -> int 
  - apply(square,[2,3,4]); 
  val it = [4,9,16] : int list
  ```