More Curried Functions

```
- hd;
> val it = fn : 'a list -> 'a

- hd [op+,op-,op*,op div] (5,4);
> val it = 9 : int
```

Here the type of `hd` is:

```
(int*int -> int) list -> int*int -> int
```

An analogy can be made with nested arrays, as in Pascal:

```
A: array [1..10] of
    array [1..10] of real
```

. . .A[i][j]. . .

Generic Sorting

```
fun insort lessequal =
 let fun ins (x,[])  = [x]
       | ins (x,h::t)=
          if lessequal(x,h) then x::h::t
          else h::ins(x,t)
     fun sort [] = []
       | sort (x::l) = ins(x,sort l)
  in sort end;

> val insort = fn :
  ('a * 'a -> bool) ->
     ('a list -> 'a list)

- insort (op<=) [5,3,5,7,2,9];
> val it = [2, 3, 5, 5, 7, 9] : int list
- insort (op>=) [5,3,5,7,2,9];
> val it = [9, 7, 5, 5, 3, 2] : int list
```

A Summation Functional

```
fun sum f 0 = 0.0
  | sum f m = f(m-1) + sum f (m-1);

> val sum =
  fn : (int -> real) -> int -> real
```

$$\texttt{sum f m} = \sum_{i=0}^{m-1} f(i)$$

$$\texttt{sum (sum f) m} = \sum_{i=0}^{m-1} \sum_{j=0}^{i-1} f(j)$$

Matrix Transpose

The `map` functional applies a function to every element of a list

```
fun map f []     = []
  | map f (h::t) = (f h)::(map f t);
```

Representing a matrix as a list of lists, the following defines the transpose function.

```
fun transp ([]::_) = []
  | transp rows    =
              (map hd rows)::
              (transp (map tl rows));
```

```
fn : 'a list list -> 'a list list
```

Matrix Multiplication

The dot product of two vectors as a curried function:

```
fun dotprod [] []             = 0.0
  | dotprod (h1::t1) (h2::t2) =
     h1*h2 + dotprod t1 t2;
```

Matrix multiplication:

```
fun matmult (Arows, Brows) =
 let val cols = transp Brows
 in map (fn row => map (dotprod row) cols)
      Arows
 end;
```

The Fold Functional

`foldl` and `foldr` are built-in functionals which can be defined as:

```
fun foldl f e []   = e
  | foldl f e (h::t) =
          foldl f f(e,h) t;

fun foldr f e []     = e
  | foldr f e (h::t) =
         f(h, foldr f e t);
```

These can be used to give simple definitions of many list functions

`foldl op+ 0`	sum
`foldl (fn (_,n) => n+1) 0`	length
`foldr op:: xs ys`	`ys@xs`

Predicates

```
fun exists p []      = false
  | exists p (h::t) = (p h) orelse
                          exists p t;
```

```
 fn : ('a -> bool) -> 'a list -> bool
```

Determines whether there is any element in a list that satisfies the predicate p.

```
fun filter p []      = []
  | filter p (h::t) = if p h then
                            h::(filter p t)
                         else filter p t;
```

```
 fn : ('a -> bool) -> 'a list -> 'a list
```