```
(* A polymorphic object that is not a function *)
val it = [] : 'a list
- 3::[];
val it = [3] : int list
- 3.5::[];
val it = [3.5] : real list

(* The identify function - no other function (that alwaysterminates)
   has type 'a -> 'a *)
- fun id x = x;
val id = fn : 'a -> 'a

(* Here's a function of type 'a -> 'a, but never terminates *)
- fun silly x  = if false then x else silly x;
val silly = fn : 'a -> 'a

(* Here's the definition of map. Note its type *)
- fun map f [] = []
=   |  map f (x::xs) = f x :: map f xs ;
val map = fn : ('a -> 'b) -> 'a list -> 'b list

(* Note that both branches of a conditional must have the same type *)
- fun foo f g x = if x = 0 then f else g ;
val foo = fn : 'a -> 'a -> int -> 'a

(* ML is a functional language, so of course you can pass
   functions as parameters *)
- fun add1 x = x+1;
val add1 = fn : int -> int
- fun add2 x = x+2;
val add2 = fn : int -> int
- foo add1 add2 3;
val it = fn : int -> int


- (* lambda expression is written as fn arg => exp *)
- fn x => x+3;
val it = fn : int -> int
- (foo (fn x => x*2) (fn x => x*4) 0) 7;
val it = 14 : int

- (* how to write composition *)
- fun compose f g = fn x => f (g x);
val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
- fun f x y = x+y;
val f = fn : int -> int -> int

- (* above is equivalent to: *)
- fun g x = fn y => x+y;
val g = fn : int -> int -> int
- val h = f 3;
val h = fn : int -> int
- h 7;
val it = 10 : int

(* tuples *)
- (3,4.5)   (* this is the tuple containing 3 and 4.5 *)
;
val it = (3,4.5) : int * real
```

```
(* writing a function that takes a tuple as a parameter *)
- fun bar (a,b) = a+b;
val bar = fn : int * int -> int
- bar (3,5);
val it = 8 : int

(* giving a name to an existing type (not creating a new type) *)
- type mytype = (int * real) -> int;
type mytype = int * real -> int
- fun myid (f: mytype) = f;
val myid = fn : mytype -> mytype
- myid (fn (x,y) => 3);
val it = fn : mytype
(* This won't work *)
- myid (fn x => x);
stdIn:36.1-36.17 Error: operator and operand don't agree [tycon mismatch]
  operator domain: mytype
  operand:         int * real -> int * real
  in expression:
    myid (fn x => x)

(* introduce a new type by using the "datatype" facility *)
- datatype stoplight = read | green | yellow;
datatype stoplight = green | read | yellow
- ;
- green;
val it = green : stoplight

(* Can use the literals of the new type in patterns *)
- fun drive green = "go"
=   |  drive read = "stop"
=   |  drive yellow = "go faster";
val drive = fn : stoplight -> string
- drive yellow;
val it = "go faster" : string

(* the alternatives in a datatype declaration can be "value constructors" *)
- datatype tree = leaf of int | node of tree * tree;
datatype tree = leaf of int | node of tree * tree

(* Here leaf is not itself a tree, but rather something that takes an int and
   creates a tree *)
- leaf 3;
val it = leaf 3 : tree
- node (leaf 4, leaf 5);
val it = node (leaf 4,leaf 5) : tree

(* using the value constructors as patterns *)
-  fun addtree (leaf x) = x
=    |  addtree (node (left, right)) = addtree left + addtree right;
val addtree = fn : tree -> int
- val t = node(node(leaf 4, node (leaf 5, leaf 6)), leaf 7);
val t = node (node (leaf #,node #),leaf 7) : tree
- addtree;
val it = fn : tree -> int
- addtree t;
val it = 22 : int

(* Computing the fringe of a tree - a list of the values at to leaves *)
- fun fringe (leaf x) = [x]
=   |  fringe (node (left, right)) = fringe left @ fringe right; (* @ is append *)
```

```
val fringe = fn : tree -> int list
- fringe t;
val it = [4,5,6,7] : int list

(* Can create "type constructors" using the datatype feature *)
- datatype 'a tree = leaf of 'a | node of ('a tree * 'a tree);
datatype 'a tree = leaf of 'a | node of 'a tree * 'a tree
(* In the above, tree is not a type -- rather it is a type constructor that
   given a type (for 'a), constructs a type *)

(* The compiler can infer the actual tree type from tree values *)
- leaf 3;
val it = leaf 3 : int tree

(* Easy to write polymorphic functions over tree types *)
- fun fringe (leaf x) = [x]
 |  fringe (node (left, right)) = fringe left @ fringe right;
= val fringe = fn : 'a tree -> 'a list
- fringe (node(leaf [1,2], leaf [3,4]));
val it = [[1,2],[3,4]] : int list list

- (* defining mutally recursive functions, use "and" (which is not the
     logical operation) *)
- fun f 0 = 1
=  |  f x = x * g (x-1)
= and
=     g 0 = 1
=  |  g y = y * f (y-1);
val f = fn : int -> int
val g = fn : int -> int
- f 5;
val it = 120 : int
-
```