(* SCOPING *)

```
val x = 1;
fun addX y = x+y;
val x = 2;
addX 0; (* = 1 *)
```

(* CURRYING *)

```
fun someCurriedFunction x y = 0;
(* 'a −> 'b −> int *)
someCurriedFunction 1 "hi";
(* int *)
```

(* HIGHER−ORDER FUNCTIONS *)

```
null;
map; (* 'a → 'a list *)
val null_mask = map null;
val xss = [[], [1,2,3]];
map null xss; (* 'a list list −> bool list *)
```

(* TAIL RECURSION *)

```
(* A "tail call" is, conceptually a function call inside another function, where
the caller just returns the callee’s return value without doing any more work.
*)
```

(* An expression is "in tail position" iff any:
* it is the ENTIRE right-hand-side of a function body
* it is the ENTIRE then-branch or else-branch of a conditional that’s in tail position
* it is the ENTIRE body of a let-expression in tail position
...I think that’s it. *)

(* e.g.*

```
datatype json = Object of (string × int) list | OmittedJsonConstructors;
fun one_fields j =
  let
    fun loop (fs,acc) =
      case fs of
        [] ⇒ acc
        | fs ⇒ "hi" :: loop([], [], would NOT be tail-recursive!) in
    in
      case j of
        Object fs ⇒ loop (fs,[])
        _ ⇒ [] end
  end

(* DATA TYPES *)

```

(* one−of *)

```
datatype Rank = Num of int | Jack | Queen | King | Ace;
```

(* self−reference *)

```
datatype Tree = Leaf of int |
  Node of (int × Tree × Tree);
```

(* TYPECHECKING *)

```
(* find : ('a → bool) −> 'a list −> 'a option
T2 = T2c list
x : T2c
xs' : T2c list
T1 = T2c → bool
T3 = T2c option
T3 = T2c option
T3 = T3
*)
fun find predicate xs = case xs of
  [] ⇒ NONE
  | x::xs' ⇒ if predicate x
      then SOME x
      else find predicate xs;
```

(* STRUCTURES AND SIGNATURES *)

```
(* structures lump together associated functionality *)
(* signatures define interfaces; structures can implement signatures *)
(* All structures implementing the same signature are ~indistinguishable *)
```

```
signature FRACTIONS =
sig
  type Fractional;
  val Whole : int → Fractional;
  val add : (Fractional × Fractional) → Fractional;
end
```

```
structure Fractions :> FRACTIONS =
  struct
    datatype Fractional = Whole of int
      | Frac of int × int;
    fun add (x, y) =
      case (x, y) of
        (Whole a, Whole b) ⇒ Whole (a+b)
        | (Whole a, Frac (n,d)) ⇒ Frac (n*a+d, d)
        | (Frac (n1, d1), Frac (n2, d2)) ⇒ Frac (n1*d2+n2*d1, d1*d2)
        | _ ⇒ add (y, x)
      val not_in_signature = 0;
  end
```

```
(* one−of *)
```

```
type person = (string × int);
```

```
datatype json = Object of (string × int) list | OmittedJsonConstructors;
fun one_fields j =
  let
    fun loop (fs,acc) =
      case fs of
        [] ⇒ acc
        | fs ⇒ "hi" :: loop([], [], would NOT be tail-recursive!) in
    in
      case j of
        Object fs ⇒ loop (fs,[])
        _ ⇒ [] end
  end
```

```
(* each−of *)
```

```
type person = (string × int);
```

```
fun listify x = [x]; (* 'a −> 'a list *)
listify "hi"; (* = ["hi"], not [0] *)
```

```
fun null; (* 'a list −> bool *)
map; (* ('a −> 'b) −> 'a list −> 'b list *)
val null_mask = map null;
val xss = [[], [1,2,3]];
map null xss; (* 'a list list −> bool list *)
```

```
val xss = [[], [1,2,3]];
map null xss; (* 'a list list −> bool list *)
```

```
val null_mask = map null;
val xss = [[], [1,2,3]];
map null xss; (* 'a list list −> bool list *)
```

```
fun null; (* 'a list −> bool *)
map; (* ('a −> 'b) −> 'a list −> 'b list *)
val null_mask = map null;
val xss = [[], [1,2,3]];
map null xss; (* 'a list list −> bool list *)
```

```
fun listify x = [x]; (* 'a −> 'a list *)
listify "hi"; (* = ["hi"], not [0] *)
```

```
fun listify x = [x]; (* 'a −> 'a list *)
listify "hi"; (* = ["hi"], not [0] *)
```

```
fun listify x = [x]; (* 'a −> 'a list *)
listify "hi"; (* = ["hi"], not [0] *)
```