```
 * - Ad-hoc and parametric polymorphism.
 * - Function types.
 * - List types and tuple types (and their differences).
 * - Equality types.
 * - ML patterns and pattern-matching.
 * - Unit testing (test/EssayTests.ml).
 * - Property based testing (test/EssayTests.ml).

(* The following function adds two numbers together, its type signature is ,
int -> int -> int *)

let add (x:int) (y:int) : int = x + y ;;

(* The following function subtracts two numbers, its type signature is ,
int -> int -> int *)
let sub (x:int) (y:int) : int = x + y ;;

let mult (x:int) (y:int) : int = x * y ;;

(* The following function demonstrates a recursive function*)
(* In many ways recursive functions in functional programming are equivalent
to loops in procedural programming *)
(* A function is recursive when it calls itself*)

let rec fibonacci n = (*In Ocaml you use the 'rec' keyword*)
  match n with
  0 -
> 1 (* You usally have 1 or more base cases, this is when the function returns
 a value but it doesn't call itself *)
| 1 -> 1
| x -> fibonacci (x-1) + fibonacci (x-
2);; (*Here we have the function calling itself 'fibonacci' *)(*The function
calls itself with different argument values, or it would be stuck in an
infinite loop*)

(* This is a recursive function to find the length of a given list *)
let rec length : 'a list -> int =
  fun xs ->
  match xs with
    [] -> 0 (* Base Case *)
  | (x::rest) -> 1 + length rest;;

  (* This is a recursive function to find if there are any evens number in an
integer list *)
let rec any_evens : int list -> bool =
  fun xs ->
  match xs with
  [] -> false (* Base Case *)
  | (x::xs) -> if is_even x
                then true
               else any_evens xs ;;

(*The following function demonstrates polymorphism *)
(*A polymorphic function is a function that can have parameters of different
types *)
(*A polymorphic value doesn't have a single type, instead it can take on many
different types as needed*)

# studentId 0234;;
- : int = 0234

# studentId true;;
- : bool = true

# studentId "callum";;
- : string = "callum"

(* As you can apply this function to many types of values, it is polymorphic *
)

(*The following function demonstrates named and anonymous functions*)
(*In Ocaml there are 2 ways to define a function, either as a named function
or an anonymous function *)
(*To create an anonymous function in Ocaml we use the keyword 'fun' then we
pass the variable we want*)

(fun x y -> (* The function takes 'x' and 'y' *)
  if x > y (* If 'x' is larger than 'y' then 'x' *)
    then x
 else y ) ;; (* Else 'y' must be the larger number *)

 (*To create a named function in Ocaml we use the keyword 'let' then we pass t
he variables we want*)

 let maxNum = fun x y -
> (*This is now a named function, it does the same as the one above but is
called maxNum*)
  if x > y
  then x
  else y ;;

(* Named function to make a word plural *)
let pluralise : string -> string =
    fun string1 -> string1 ^ "s";;

(* Anonymous function to make a word plural *)
```

```
(fun string1 -> string1 ^ "s") ;;

(* The following function demonstrates function type signatures *)


# 'a -
> 'b = <fun> (* This takes some value of type 'a' and returns some value of
type 'b' *)

# 'a -> 'b -
> 'c = <fun> (* The function with this type, takes 2 values of type 'a' and 'b
' and returns a value of type 'c' *)

# (+) ;;
int -> int -
> int = <fun> (* This is the function type signature for integer addition *)

# (-) ;;
int -> int -
> int = <fun> (* This is the function type signature for integer subtraction *
)

# (+.) ;;
 float -> float -
> float = <fun> (* This is the function type signature for float addition *)

(* The following function demonstrates pattern matching in Ocaml *)

(* Pattern matching works by matching an expression with a pattern and then ap
plying or evaluating an expression *)

# let day_of_week i =
  match i with    (* This is pattern matching, we match the integer 'i' (expre
ssion) *)
   1 -
> "Monday"  (* with the pattern 'n' and if they match apply the experssion on
the right of the '->' *)
|  2 -> "Tuesday"
|  3 -
> "Wednesday" (* If we give the integer '3' then we match 'i' with the pattern
 3 and return "Wednesday" *)
|  4 -> "Thursday"
|  5 -> "Friday"
|  6 -> "Saturday"
|  7 -> "Sunday"

(* Another example of pattern matching, on an AND truth table *)
(* Here we match b1 and b2 with the cases in our truth table *)
let truth_table_and : bool -> bool -> bool =
  fun b1 b2 -> match b1, b2 with
  true, true -> true
 | true, false -> false
 | false, true -> false
 | false, false -> false ;;

(* In Ocaml you can read data from a list using the 'match' statement. *)

(* Here we have a recursive function that sums the elements in a list *)

let rec sum i = (* Using recursive with keyword 'rec' *)
  match i with (* Pattern matching using the 'match with' statement *)
   | [] -> 0 (* Empty list *)
   | hd :: tl -
> hd + sum tl ;; (* head to tail of the list, adding the elements recursively
*)

(* The following function demonstrates list types and tuple types *)

(* In Ocaml all elements of a linked list MUST have the same type, a linked
list of integers has the type 'int list' *)

# [1; 2; 3];;
- : int list = [1; 2; 3]

# ['a'; 'b'; 'c'];;
- : char list = ['a'; 'b'; 'c']

# ["String"; "Of"; "Lists"];;
- : string list = ["String"; "Of"; "Lists"]

# [];; (* This is an empty list *)
- : 'a list = []


(* A list has a head which is the first element, and a tail which is the rest
of the elements, as the head is only the first element *)
(* this means the head is an element and the tail is a list *)

(* [3; 4; 5; 7] - Head 3 , tail [4; 5; 7] *)

(* You can also use the cons operator to write a list '::', this is used in
pattern matching on lists *)
(* Although all elements in a linked list must have the same type, this type
can be polymorphic allowing you to have generic lists of ints and string etc *
)
```

```
(* There is a big difference between lists and tuples, where the elements of a
 list must all have the same type, a tuple can contain elements of different
types *)

(* (1,3) - This is a tuple of type 'int * int' *)

# (1,3);;
- : int * int = (1, 3)

(* ("word", 5, 'a') - This is a tuple of type ' string * int * char ' *)

# ("word", 5, 'a');;
- : string * int * char = ("word", 5, 'a')
```