OCaml is a strictly evaluated functional language with some imperative features.

Along with StandardML and its dialects it belongs to ML language family. F# is also heavily influenced by OCaml.

Just like StandardML, OCaml features both an interpreter, that can be used interactively, and a compiler. The interpreter binary is normally called "ocaml" and the compiler is "ocamlopt". There is also a bytecode compiler, "ocamlc", but there are few reasons to use it.

It is strongly and statically typed, but instead of using manually written type annotations, it infers types of expressions using Hindley-Milner algorithm. It makes type annotations unnecessary in most cases, but can be a major source of confusion for beginners.

When you are in the top level loop, OCaml will print the inferred type after you enter an expression.

```
# let inc x = x + 1 ;;
val inc : int -> int = <fun>
# let a = 99 ;;
val a : int = 99
```

For a source file you can use "ocamlc -i /path/to/file.ml" command to print all names and type signatures.

```
$ cat sigtest.ml
let inc x = x + 1
let add x y = x + y

let a = 1

$ ocamlc -i ./sigtest.ml
val inc : int -> int
val add : int -> int -> int
val a : int
```

Note that type signatures of functions of multiple arguments are written in curried form. A function that takes multiple arguments can be represented as a composition of functions that take only one argument. The "f(x,y) = x + y" function from the example above applied to arguments 2 and 3 is equivalent to the "f0(y) = 2 + y" function applied to 3. Hence the "int -> int -> int" signature.

```
(*** Comments ***)

(* Comments are enclosed in (* and *). It's fine to nest comments. *)

(* There are no single-line comments. *)


(*** Variables and functions ***)

(* Expressions can be separated by a double semicolon symbol, ";;".
   In many cases it's redundant, but in this tutorial we use it after
   every expression for easy pasting into the interpreter shell.
   Unnecessary use of expression separators in source code files
   is often considered to be a bad style. *)

(* Variable and function declarations use "let" keyword. *)
let x = 10 ;;
```

```
(* OCaml allows single quote characters in identifiers.
   Single quote doesn't have a special meaning in this case, it's often used
   in cases when in other languages one would use names like "foo_tmp". *)
let foo = 1 ;;
let foo' = foo * 2 ;;

(* Since OCaml compiler infers types automatically, you normally don't need to
   specify argument types explicitly. However, you can do it if
   you want or need to. *)
let inc_int (x: int) : int = x + 1 ;;

(* One of the cases when explicit type annotations may be needed is
   resolving ambiguity between two record types that have fields with
   the same name. The alternative is to encapsulate those types in
   modules, but both topics are a bit out of scope of this
   tutorial. *)

(* You need to mark recursive function definitions as such with "rec" keyword. *)
let rec factorial n =
    if n = 0 then 1
    else n * factorial (n-1)
;;

(* Function application usually doesn't need parentheses around arguments *)
let fact_5 = factorial 5 ;;

(* ...unless the argument is an expression. *)
let fact_4 = factorial (5-1) ;;
let sqr2 = sqr (-2) ;;

(* Every function must have at least one argument.
   Since some funcions naturally don't take any arguments, there's
   "unit" type for it that has the only one value written as "()" *)
let print_hello () = print_endline "hello world" ;;

(* Note that you must specify "()" as argument when calling it. *)
print_hello () ;;

(* Calling a function with insufficient number of arguments
   does not cause an error, it produces a new function. *)
let make_inc x y = x + y ;; (* make_inc is int -> int -> int *)
let inc_2 = make_inc 2 ;;   (* inc_2 is int -> int *)
inc_2 3 ;; (* Evaluates to 5 *)

(* You can use multiple expressions in function body.
   The last expression becomes the return value. All other
   expressions must be of the "unit" type.
   This is useful when writing in imperative style, the simplest
   form of it is inserting a debug print. *)
let print_and_return x =
    print_endline (string_of_int x);
    x
;;
```

```
(* Since OCaml is a functional language, it lacks "procedures".
   Every function must return something. So functions that
   do not really return anything and are called solely for their
   side effects, like print_endline, return value of "unit" type. *)


(* Definitions can be chained with "let ... in" construct.
   This is roughly the same to assigning values to multiple
   variables before using them in expressions in imperative
   languages. *)
let x = 10 in
let y = 20 in
x + y ;;

(* Alternatively you can use "let ... and ... in" construct.
   This is especially useful for mutually recursive functions,
   with ordinary "let .. in" the compiler will complain about
   unbound values. *)
let rec
  is_even = function
  | 0 -> true
  | n -> is_odd (n-1)
and
  is_odd = function
  | 0 -> false
  | n -> is_even (n-1)
;;

(* Anonymous functions use the following syntax: *)
let my_lambda = fun x -> x * x ;;

(*** Operators ***)

(* There is little distintion between operators and functions.
   Every operator can be called as a function. *)

(+) 3 4  (* Same as 3 + 4 *)

(* There's a number of built-in operators. One unusual feature is
   that OCaml doesn't just refrain from any implicit conversions
   between integers and floats, it also uses different operators
   for floats. *)
12 + 3 ;; (* Integer addition. *)
12.0 +. 3.0 ;; (* Floating point addition. *)

12 / 3 ;; (* Integer division. *)
12.0 /. 3.0 ;; (* Floating point division. *)
5 mod 2 ;; (* Remainder. *)

(* Unary minus is a notable exception, it's polymorphic.
   However, it also has "pure" integer and float forms. *)
- 3 ;; (* Polymorphic, integer *)
- 4.5 ;; (* Polymorphic, float *)
```

```
~- 3 (* Integer only *)
~- 3.4 (* Type error *)
~-. 3.4 (* Float only *)

(* You can define your own operators or redefine existing ones.
   Unlike SML or Haskell, only selected symbols can be used
   for operator names and first symbol defines associativity
   and precedence rules. *)
let (+) a b = a - b ;; (* Surprise maintenance programmers. *)

(* More useful: a reciprocal operator for floats.
   Unary operators must start with "~". *)
let (~/) x = 1.0 /. x ;;
~/4.0 (* = 0.25 *)


(*** Built-in data structures ***)

(* Lists are enclosed in square brackets, items are separated by
   semicolons. *)
let my_list = [1; 2; 3] ;;

(* Tuples are (optionally) enclosed in parentheses, items are separated
   by commas. *)
let first_tuple = 3, 4 ;; (* Has type "int * int". *)
let second_tuple = (4, 5) ;;

(* Corollary: if you try to separate list items by commas, you get a list
   with a tuple inside, probably not what you want. *)
let bad_list = [1, 2] ;; (* Becomes [(1, 2)] *)

(* You can access individual list items with the List.nth function. *)
List.nth my_list 1 ;;

(* There are higher-order functions for lists such as map and filter. *)
List.map (fun x -> x * 2) [1; 2; 3] ;;
List.filter (fun x -> if x mod 2 = 0 then true else false) [1; 2; 3; 4] ;;

(* You can add an item to the beginning of a list with the "::" constructor
   often referred to as "cons". *)
1 :: [2; 3] ;; (* Gives [1; 2; 3] *)

(* Arrays are enclosed in [| |] *)
let my_array = [| 1; 2; 3 |] ;;

(* You can access array items like this: *)
my_array.(0) ;;


(*** Strings and characters ***)

(* Use double quotes for string literals. *)
let my_str = "Hello world" ;;
```

```
(* Use single quotes for character literals. *)
let my_char = 'a' ;;

(* Single and double quotes are not interchangeable. *)
let bad_str = 'syntax error' ;; (* Syntax error. *)

(* This will give you a single character string, not a character. *)
let single_char_str = "w" ;;

(* Strings can be concatenated with the "^" operator. *)
let some_str = "hello" ^ "world" ;;

(* Strings are not arrays of characters.
   You can't mix characters and strings in expressions.
   You can convert a character to a string with "String.make 1 my_char".
   There are more convenient functions for this purpose in additional
   libraries such as Core.Std that may not be installed and/or loaded
   by default. *)
let ocaml = (String.make 1 'O') ^ "Caml" ;;

(* There is a printf function. *)
Printf.printf "%d %s" 99 "bottles of beer" ;;

(* Unformatted read and write functions are there too. *)
print_string "hello world\n" ;;
print_endline "hello world" ;;
let line = read_line () ;;


(*** User-defined data types ***)

(* You can define types with the "type some_type =" construct. Like in this
   useless type alias: *)
type my_int = int ;;

(* More interesting types include so called type constructors.
   Constructors must start with a capital letter. *)
type ml = OCaml | StandardML ;;
let lang = OCaml ;;  (* Has type "ml". *)

(* Type constructors don't need to be empty. *)
type my_number = PlusInfinity | MinusInfinity | Real of float ;;
let r0 = Real (-3.4) ;; (* Has type "my_number". *)

(* Can be used to implement polymorphic arithmetics. *)
type number = Int of int | Float of float ;;

(* Point on a plane, essentially a type-constrained tuple *)
type point2d = Point of float * float ;;
let my_point = Point (2.0, 3.0) ;;

(* Types can be parameterized, like in this type for "list of lists
   of anything". 'a can be substituted with any type. *)
type 'a list_of_lists = 'a list list ;;
```

```
type int_list_list = int list_of_lists ;;

(* Types can also be recursive. Like in this type analogous to
   built-in list of integers. *)
type my_int_list = EmptyList | IntList of int * my_int_list ;;
let l = IntList (1, EmptyList) ;;


(*** Pattern matching ***)

(* Pattern matching is somewhat similar to switch statement in imperative
   languages, but offers a lot more expressive power.

   Even though it may look complicated, it really boils down to matching
   an argument against an exact value, a predicate, or a type constructor.
   The type system is what makes it so powerful. *)

(** Matching exact values.  **)

let is_zero x =
    match x with
    | 0 -> true
    | _ -> false  (* The "_" pattern means "anything else". *)
;;

(* Alternatively, you can use the "function" keyword. *)
let is_one = function
| 1 -> true
| _ -> false
;;

(* Matching predicates, aka "guarded pattern matching". *)
let abs x =
    match x with
    | x when x < 0 -> -x
    | _ -> x
;;

abs 5 ;; (* 5 *)
abs (-5) (* 5 again *)

(** Matching type constructors **)

type animal = Dog of string | Cat of string ;;

let say x =
    match x with
    | Dog x -> x ^ " says woof"
    | Cat x -> x ^ " says meow"
;;

say (Cat "Fluffy") ;; (* "Fluffy says meow". *)

(** Traversing data structures with pattern matching **)
```

```
(* Recursive types can be traversed with pattern matching easily.
   Let's see how we can traverse a data structure of the built-in list type.
   Even though the built-in cons ("::") looks like an infix operator,
   it's actually a type constructor and can be matched like any other. *)
let rec sum_list l =
    match l with
    | [] -> 0
    | head :: tail -> head + (sum_list tail)
;;

sum_list [1; 2; 3] ;; (* Evaluates to 6 *)

(* Built-in syntax for cons obscures the structure a bit, so we'll make
   our own list for demonstration. *)

type int_list = Nil | Cons of int * int_list ;;
let rec sum_int_list l =
  match l with
      | Nil -> 0
      | Cons (head, tail) -> head + (sum_int_list tail)
;;

let t = Cons (1, Cons (2, Cons (3, Nil))) ;;
sum_int_list t ;;
```

Further reading

- Visit the official website to get the compiler and read the docs: http://ocaml.org/
- Try interactive tutorials and a web-based interpreter by OCaml Pro: http://try.ocamlpro.com/
- Read "OCaml for the skeptical" course: http://www2.lib.uchicago.edu/keith/ocaml-class/home.html