OCaml Syntax Cheatsheet

Basic Values and Types

```
(* Values *)
42                  (* int *)
3.14               (* float *)
"hello"            (* string *)
'a'                (* char *)
true               (* bool *)
()                 (* unit *)
[||]               (* empty array *)
None               (* option *)

(* Type Annotations *)
let x: int = 42
let f (x: int) : string = string_of_int x
let g : int -> int = fun x -> x + 1
```

Variables and Functions

```
(* Let Bindings *)
let x = 42
let y = x + 1
let z =
  let temp = 10 in
  temp * 2

(* Functions *)
let add x y = x + y
let add2 = fun x y -> x + y
let add3 = (+)  (* operator as function *)

(* Recursive Functions *)
let rec factorial n =
  if n = 0 then 1
  else n * factorial (n - 1)

(* Mutual Recursion *)
let rec even n =
  if n = 0 then true
  else odd (n-1)
and odd n =
  if n = 0 then false
  else even (n-1)
```

Pattern Matching

```
(* Basic Match with Options *)
let extract_value = function
  | Some x -> x
  | None -> 0

(* Match with Lists *)
let describe_list = function
  | [] -> "empty"
  | [x] -> "singleton"
  | x::xs -> "non-empty"

(* Match with Tuples *)
let describe_point = function
  | (0, 0) -> "origin"
  | (x, 0) -> "on x-axis"
  | (0, y) -> "on y-axis"
  | (x, y) -> "other"

(* Guards with Complex Conditions *)
let grade score =
  match score with
  | n when n >= 90 -> "A"
  | n when n >= 80 -> "B"
  | n when n >= 70 -> "C"
  | n when n >= 60 -> "D"
  | _ -> "F"

(* Guards with Multiple Patterns *)
let describe_number n =
  match n with
  | 0 -> "zero"
  | n when n mod 2 = 0 -> "even"
  | n when n > 100 -> "large odd"
  | _ -> "small odd"

(* Pattern Matching with Variants *)
type payment = Cash | Card of string | Check of int
let process_payment = function
  | Cash -> "processing cash"
  | Card num when String.length num = 16 -> "processing card"
  | Card _ -> "invalid card"
  | Check n when n > 0 -> "processing check"
  | Check _ -> "invalid check"
```

Data Structures

```
(* Lists *)
let empty = []
```

```
let numbers = [1; 2; 3]
let combined = 0 :: numbers
1 :: 2 :: 3 :: []     (* cons operator *)
List.hd numbers       (* head *)
List.tl numbers       (* tail *)
List.length numbers   (* length *)
let concat = [1;2] @ [3;4]  (* concatenation *)

(* Tuples *)
let pair = (1, "one")
let triple = (1, "one", true)
let (x, y) = pair     (* destructuring *)

(* Records *)
type person = {
  name: string;
  age: int;
  mutable location: string;  (* mutable field *)
}
let bob = { name = "Bob"; age = 25; location = "NY" }
let name = bob.name   (* field access *)
bob.location <- "LA"  (* mutable field update *)

(* Variants *)
type shape =
  | Circle of float
  | Rectangle of float * float
let area = function
  | Circle r -> 3.14 *. r *. r
  | Rectangle (w, h) -> w *. h
```

Options and Results

```
(* Option Type *)
type 'a option = None | Some of 'a
let safe_div x y =
  if y = 0 then None
  else Some (x / y)

(* Result Type *)
type ('a, 'e) result = Ok of 'a | Error of 'e
let safe_div x y =
  if y = 0 then Error "Division by zero"
  else Ok (x / y)
```

Modules and Functors

```
(* Module Definition *)
module Math = struct
```

```
  let add x y = x + y
  let sub x y = x - y
end

(* Module Signature *)
module type MATH = sig
  val add : int -> int -> int
  val sub : int -> int -> int
end

(* Module with Signature *)
module SafeMath : MATH = Math

(* Module Usage *)
let sum = Math.add 2 3
let open Math in add 2 3  (* local open *)

(* Functor Definition *)
module type Comparable = sig
  type t
  val compare : t -> t -> int
end

module MakeSet (Item: Comparable) = struct
  type elt = Item.t
  type t = elt list
  (* ... *)
end
```

Operators

```
(* Arithmetic *)
+    (* int addition *)
+.   (* float addition *)
-    (* int subtraction *)
-.   (* float subtraction *)
*    (* int multiplication *)
*.   (* float multiplication *)
/    (* int division *)
/.   (* float division *)

(* Comparison *)
=    (* structural equality *)
==   (* physical equality *)
<>   (* structural inequality *)
!=   (* physical inequality *)
<    (* less than *)
<=   (* less than or equal *)
>    (* greater than *)
>=   (* greater than or equal *)
```

```
(* Boolean *)
&&   (* and *)
||   (* or *)
not  (* not *)

(* List *)
@    (* list concatenation *)
::   (* cons operator *)
```