OCaml Cheat Sheet

A concise reference for OCaml syntax, data structures, and common functions. Perfect for quick lookups and reminders.

Basics & Syntax

Basic Syntax

Variable Declaration	`let variable_name = value;;` `let variable_name = value in expression`
Function Definition	`let function_name argument1 argument2 = expression;;`
Function Application	`function_name argument1 argument2` (no parentheses needed)
Semicolon Usage	`;;` terminates top-level phrases in the interactive toplevel. `;` sequences expressions, like `expr1; expr2`
Comments	`(* This is a comment *)`
If-Then-Else	`if condition then expression1 else expression2`

Basic Data Types

Integer	`10`, `-5`, `0`
Float	`3.14`, `-2.0`, `0.0` (use `+.`, `-.`, `*.`, `/.` for operations)
Boolean	`true`, `false`
Character	`'a'`, `'Z'`, `'\n'`
String	`"hello"`, `"world"` (immutable)
Unit	`()` (used for side-effects)

Operators

Arithmetic (int)	`+`, `-`, `*`, `/`, `mod`
Arithmetic (float)	`+.`, `-.`, `*.`, `/.`, `**`
Comparison	`=`, `<>`, `<`, `>`, `<=`, `>=` (structural equality)
Logical	`&&`, `||`, `not`
String Concatenation	`^`

Data Structures

Tuples

Definition	`let my_tuple = (1, "hello", true);;`
Accessing Elements	`let (a, b, c) = my_tuple in a;;` `fst (1,2);;` (* returns 1 *) `snd (1,2);;` (* returns 2 *)
Type	`int * string * bool`

Lists

Definition	`let my_list = [1; 2; 3; 4];;` `let empty_list = [];;`
Cons Operator	`1 :: [2; 3];;` (* Result: `[1; 2; 3]` *)
Head and Tail	`List.hd [1; 2; 3];;` (* returns 1 *) `List.tl [1; 2; 3];;` (* returns [2;3] *)
Common Functions	`List.length`, `List.map`, `List.filter`, `List.fold_left`
Type	`int list`, `string list`

Arrays

Definition	`let my_array = [| 1; 2; 3; 4 |];;`
Accessing Elements	`my_array.(0);;` (*Access first element*)
Modifying Elements	`my_array.(0) <- 5;;`
Type	`int array`, `string array`

Records

Definition	`type person = { name : string; age : int };;` `let john = { name = "John"; age = 30 };;`
Accessing Fields	`john.name;;` (* returns "John" *)
Modifying Fields (using `with` keyword)	`let older_john = { john with age = john.age + 1 };;`

Variants

Definition	`type color = Red | Green | Blue;;` `let my_color = Red;;`

Parameterized Variants	`type option = Some of 'a | None;;`
	`let some_value = Some 10;;`
	`let no_value = None;;`

Recursive Variants	`type int_tree = Leaf of int | Node of int_tree * int_tree;;` `let my_tree = Node (Leaf 1, Leaf 2);;`

Functions & Modules

Function Basics

Anonymous Functions	`fun x -> x + 1;;`
Currying	All functions in OCaml are curried by default. `let add x y = x + y;;` `let add_five = add 5;;` (* `add_five` is a function that adds 5 to its argument *)
Recursive Functions	`let rec factorial n = if n <= 1 then 1 else n * factorial (n - 1);;`

Higher-Order Functions

Map	`List.map (fun x -> x * 2) [1; 2; 3];;` (* Result: `[2; 4; 6]` *)
Filter	`List.filter (fun x -> x mod 2 = 0) [1; 2; 3; 4];;` (* Result: `[2; 4]` *)
Fold Left	`List.fold_left (fun acc x -> acc + x) 0 [1; 2; 3];;` (* Result: `6` *)
Fold Right	`List.fold_right (fun x acc -> acc + x)  [1; 2; 3] 0;;` (* Result: `6` *)

Modules

Module Definition	`module MyModule = struct let x = 10 let add y = x + y end;;`
Module Usage	`MyModule.x;;` `MyModule.add 5;;`
Module Types (Interfaces)	`module type MY_MODULE_TYPE = sig val x : int val add : int -> int end;;`
Module Implementation	`module MyModule : MY_MODULE_TYPE = struct let x = 10 let add y = x + y end;;`

Functors

Functor Definition	`module type Comparable = sig type t val compare : t -> t -> int end;;`
	
	`module MakeSet (C : Comparable) = struct (* Set implementation using C.t for elements *) end;;`

Functor Application	`module IntComparable = struct type t = int let compare = Stdlib.compare end;;`
	
	`module IntSet = MakeSet (IntComparable);;`

Advanced Features

Pattern Matching

Basic Matching	`let rec describe_list lst = match lst with | [] -> "empty" | [x] -> "singleton" | _ -> "longer";;`
Matching with Guards	`let classify_number x = match x with | x when x > 0 -> "positive" | x when x < 0 -> "negative" | _ -> "zero";;`
Matching on Tuples	`let process_tuple t = match t with | (1, "hello") -> "Special tuple" | (a, b) -> "Generic tuple";;`

Exceptions

Defining Exceptions	`exception MyException of string;;`
Raising Exceptions	`raise (MyException "Something went wrong");;`
Handling Exceptions	`try (* Code that may raise an exception *) with | MyException msg -> (* Handle the exception *)`

References

Creating References	`let my_ref = ref 0;;`
Accessing Reference Value	`!my_ref;;`
Modifying Reference Value	`my_ref := 5;;`

Objects

Basic Object Definition	`class point x_init y_init = object (self) val mutable x = x_init val mutable y = y_init method get_x = x method get_y = y method move dx dy = x <- x + dx; y <- y + dy end;;`
Creating an Object	`let my_point = new point 1 2;;`
Using Object Methods	`my_point#get_x;;` `my_point#move 3 4;;`