OCaml Cheat Sheet

OCaml is a strict language; it is strongly typed where types are inferred.
I may write explicit type annotations below for demonstration or clarity purposes.

```
(* Using explicit type annotations *)
let x : int = 3;;
let f (x : int) (y : string) (r : 'a) : float = 3.14;;
```

Only when interacting with the top-level interpreter, commands must be terminated by `;;`. OCaml uses `;` as an expression *separator* —not a terminator!
My Emacs setup for OCaml can be found on this CheatSheet's repo.

Arithmetic

Operations on floats have a '.' suffix.

```
8 / 3;;        (* 2 *)
8 /. 3;;       (* Type error: /. is for floats *)
8.0 /. 3.0;;   (* 2.6666... *)
```

Functions & Variables

A function is declared with the `let` keyword —variables are functions of zero arguments.

Function & varaible names *must* begin with a lowercase letter, and may use _ or '.

- They cannot begin with capital letters or numbers, or contain dashes!
- Functions are like variables, but with arguments, so the same syntax applies.

```
(* A curried function *)
let f x y = x + y

(* Function application *)
let result = f 10 (2 * 6)

(* Partial application *)
let g x = f x 2
```

```
(* We can re-bind variables *)
let x = 123
let x = string_of_int x
```

Recursive functions are marked with the `rec` keyword.

```
let rec fact n = if n = 0
                 then 1
                 else n * fact (n - 1)
```

Here's an example of a higher-order function & multiple local functions & an infix operator & an anonymous function & the main method is parametricly polymorphic. that contains & a local function & anonymous function.

```
let try_add (bop : 'a -> 'a -> 'a) (test : 'a -> bool)
            (default : 'a) (x : 'a) (y : 'a)
  = let (/@/) x y = bop x y
    (* Only select symbols can be used as infix operators *)
    (* (/@/) x y  =  x /@/ y *)
    and wrap a = if test a then a else default
    in wrap x /@/ wrap y;;

 699 = try_add (+) (fun a -> a mod 3 = 0) (666) (~-1) 33;;
 (* The anonymouse function uses '=' as Boolean equality. *)

 ~- 2 = ~- 2 mod 3;; (* /Remainder/ after dividing out 3s *)
```

```
(* Unit type; usage: my_io () *)
let my_io () = print_endline "Hello World!" ;;
```

OCaml is a functional language: *Procedures* are functions returning the unit type.
A *function* is a sequence of expressions; its *return value* is the value of the final expression —all other expressions are of unit type.

```
let const x y
  = my_io();
    y;
    x

let res = const 1972 12
```

Booleans

Inequality is expressed with `<>`.

```
(* false, true, false, true, false, true, true, 1 *)
true = false , true || false, true && false, true >= false
, 12 < 2, "abc" <= "abd", 1 <> 2
, if true then 1 else 2
```

Strings

OCaml strings are not arrays, or lists, of characters as in C or Haskell.

```
"string catenation" = "string " ^ "catenation"

Printf.printf "%d %s" 1972 "taxi";;
let input = read_line ();;
```

Records

Records: Products with named, rather than positional, components.

```
type point2d = {x : float; y : float};;
let p = {y = 2.0; x = 3.4};; (* Construction *)
let {x = px; y = py} = p;; (* Pattern matching for deconstruction *)
let go {x = qx; y = qy} = qx +. qy;;
(* More tersely, using "field pruning": Variables must coincide with field names.
let erroenous ({xx; y} : point2d )= x +. y;;
let works {x; y} = 0.0;;
(* Or we can use dot notation *)
let go q = q.x +. q.y;;
```

Variants and Pattern Matching

Variant types: A unified way to combine different types into a single type; Each case is distinuighed by a a captialised tag.

```
type fancy_num = Boring of int | AlsoBoring of float | Fancy of point2d
```

```
(* Type alias *)
type myints = int

(* Constructors must start with a capital letter, like in Haskell *)
type 'a term = Nothing | Var of 'a | Add of 'a term * 'a term
let example = Add (Var 666, Nothing)

(* Guarded pattern matching *)
let rec sum acc = function | Nothing -> 0 + (match acc with true -> 1 | false -> 0)
                           | Var x when x <= 0 -> 0
                           | (Var 666) as p -> failwith "Evil!"
                           | Add(l, r) -> sum acc l + sum acc r
                           | _ -> 2 (* Default case *)

let res = sum true example
```

Note that we can give a pattern a name; above we mentioned `p`, but did not use it.

- Repeated & non-exhaustive patterns trigger a warning; e.g., remove the default case above.
- You can pattern match on arrays too; e.g., `[| x ; y ; z|] -> y`.

Tuples and Lists

Tuples: Parentheses are optional, comma is the main operator.

```
let mytuple  : int * string * float = (3, "three", 3.0);;
 (* Pattern matching for tuples can also be used to extract components *)
let (woah0, woah1, woah2) = mytuple;;
let add_1and4 (w, x, y, z) = w + z;;

(* Tuples: Char, String, Bool  *)
let ys = 'a', "two", true
let that = fst ("that", false)

(* A singelton list of one tuple !!!!  *)
let zs = [ 1, "two", true ]

(* Lists:  type 'a list  [] | (::) of 'a * 'a list  *)
let xs = [1; 2; 3]
[1; 2; 3] = 1 :: 2 :: 3 :: [];; (* Syntactic sugar *)

(* List catenation *)
[1;2;4;6] = [1;2] @ [4;6];;

(* Labelled arguments, using '~', means position is irrelevant in invocations *)
[1; 2; 3] = List.map ["a", "ab", "abc"] ~f:String.length;;
[1; 2; 3] = List.map  ~f:String.length ["a", "ab", "abc"];;

(* Pattern matching example; Only works on lists of length 3 *)
let go [x; y; z] = x + y + z;;
14 = go [2;5;7];;
```

Options

Option: Expressing whether a value is present or not.

```
let divide x y : int option = if y = 0 then None else Some (x / y);;
let getInt ox = match ox with None -> 0 | Some x -> x;;
0 = getInt None;;
2 = getInt (Some 2);;
```

Imperative programming —arrays

Zero-indexed Arrays: Indexing with '.(i)' and update with '<-'.

```
let nums : int array = [| 1; 2; 3 |];;
nums.(0) <- 12;;

(* Boolean tests *)
12 = xs_arr.(0)
[|12; 2; 3|] = nums;;

(* Operations whose use produce a side-effect return the 'unit' type.
 , This' akin to the role played by 'void' in C. *)
let ex : unit = ();
let myupdate (arr : 'a array) (e : 'a) (i : int) : unit = arr.(i) <- e;;
myupdate nums 33 1;;
[|12; 33; 3|] = nums;;
```

Sequencing

We may use begin/end or parentheses to group expressions together.

```
begin
  print_string "nice";
  "bye";
  true;
  10
end

;;
(  print_string "a"
 ; () (* This is the unit value *)
 ; 9
)
;;

let x = begin 1 * 2 end + (3 - 2)
```

Imperative programming —mutable records

Record fields are immutable by default, but can be declared mutable.

```
type running_sum = {mutable sum : int; mutable more : int};;
let create () = {sum = 0; more = 0};;
let update rs x =   rs.sum  <- rs.sum + rs.more
```

```
                  ; rs.more <- x;;
(* Note that ‘;’ is for Sequencing whereas ‘;;’ is for termination *)
let res = create ()
in   update res 12
   ; {sum = 0; more = 12} = res ;;
```

Refs

Refs: Single mutable values; i.e., a record with a single mutable field named ‘contents’.

```
let x : int ref = {contents = 0};;
x.contents <- x.contents + 1;;
{contents = 1} = x;;
(* These come with a handful of convenience methods: *)
```

Here’s their re-implementation:

```
(* Alias    *) type 'a ref = {mutable contents : 'a};;
(* Creation *) let ref v   = {contents = v};;
(* Access   *) let (!) r   = r.contents;; (* “value of” *)
(* Update   *) let (:=) r e = r.contents <- e;;

(* Summing the first 10 numbers *)
let sum = ref 0;;
for i = 0 to 10 do sum := !sum + i done;;
55 =  !sum;;
```

Loops

At each iteration, cons the counter i to the value of the list *so far*:

```
(* Using “i = 1 to 10” yields the reverse  *)
let xl = ref [] in
for i = 10 downto 1 do
xl := i :: !xl;
done;
!xl

let n = 100 and i = ref 0 and x = ref 0 in
while n <> !i do
  x := !x + !i; i := !i + 1;
done;
!x , 2 * !x = n * (n - 1)
```

Reads

- ☐ Learn x in y minutes, where x = OCaml
- ☐ Try OCaml, online
- ☐ Real World OCaml
- ☐ Unix system programming in OCaml
- ☐ Objective CAML Tutorial
- ☐ OCaml tutorials