Musa Al-hassy https://github.com/alhassy/OCamlCheatSheet June 3, 2019

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.

```
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 *)
```

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};;

(* Construction *)
let p = {y = 2.0; x = 3.4};;

(* Pattern matching for deconstruction *)
let {x = px; y = py} = p;;
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 capitalised tag.

```
(* Constructors must start with a capital letter, like in Haskell *)
type 'a fancy_num =   Nothing | Boring of int | Fancy of 'a
                    | Point of point2d | Pair of 'a fancy_num * 'a fancy_num

let example = Pair (Fancy "twenty", Point {x = 1.2; y = 3.14})
```

The tags allow us to *extract* components of a variant value as well as to case against values by inspecting their tags. This is *pattern matching.*

```
(* Destructuring a value *)
let Pair(head, _) = example;;
```

```
(* Guarded pattern matching, with in-line casing via ‘match’ *)
let rec sum acc = function
  | Nothing -> 0 + (match acc with true -> 1 | false -> 0)
  | Fancy x when x <= "nine" -> 0
  | (Fancy "twenty") as p -> failwith "Evil!"
  | Pair(l, r) -> sum acc l + sum acc r
  | _ -> 2 (* Default case *)

let res = sum true example (* Exception: Failure "Evil!" *)

(* Type aliases can also be formed this way *)
type myints = int
```

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`.

The above mechanisms apply to all variants —including tuples, lists, and options.

Tuples and Lists

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

```
let mytuple  : int * string * float = (3, "three", 3.0);;

(* Pattern matching & projection *)
let (woah0, woah1, woah2) = mytuple;;
let add_1and4 (w, x, y, z) = w + z;;
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];;

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

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

Options

Option: Expressing whether a value is present or not.

```
(* type 'a option = None | Some of 'a *)

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 —Sequencing & `main`

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

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

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

```
(* Begin-end block *)
begin
  print_string "nice";
  "bye";
  true;
  10
end;;
```

Remember: Single semicolon `;` is for sequencing whereas double `;;` is for termination.

OCaml programs don’t have a unique `main` function as in C, instead the entire file is evaluated sequentially at startup, which may contain arbitrary expressions not just functional declarations, and so in some sense the full codebase is one big `main` function.

Imperative Programming —Arrays

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

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

12 = nums.(0);;
[|12; 2; 3|] = nums;;
```

What is the *type* of update, `<-`? A function that returns the unit type! —see `myupdate` below.

Imperative Programming —Unit Type

Operations whose use produces a side-effect return the `unit` type. This’ akin to the role played by `void` in C. A *function* is a sequence of expressions; its *return value* is the value of the final expression —all other expressions are of unit type.

```
(* type unit = () *)
let ex : unit = ();;

let myupdate (arr : 'a array) (e : 'a)
             (i : int) : unit
  = arr.(i) <- e;;

myupdate nums 33 1;;
[|12; 33; 3|] = nums;;
```

```
let my_io () = print_endline "Hello!" ;;

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

let res = const 1972 12;;
```

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;;

let  res = create ()
in   update res 12
   ; {sum = 0; more = 12} = res ;;
```

Imperative Programming —References

References: Single mutable values; i.e., a record with a single mutable field named `contents`.

```
let x : int ref = {contents = 0}
in x.contents <- x.contents + 1; {contents = 1} = x;;
```

Refs 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;;
```

Notice that `ref` is overloaded: As a type former and as a function forming values.

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

Modules

OCaml files not only store & manage code, but also correspond to (second-class) 'modules', which act as boundaries that divide a program into conceptual units.

- ◇ At its simplest, a module is a collection of definitions that are stored within a namespace.
- ◇ Implementation details of a module can be hidden by using an *interface, module type, signature* —all are aliases.
  - ○ `val` declarations specify values in a signature: `val identifier : type`.
  - ○ A type is *abstract* if its name is exposed in the interface but its definition is not: `type t`.
    - ⋆ It is conventional to use `t` for the name of such types.
  - ○ Including the type definition in the interface makes the type *concrete*.
- ◇ E.g., module names are derived, with capitalisation, automatically from file names. An interface for `myfile.ml` to constrain exposed interface is placed in `myfile.mli`. This is nearly how C header and implementation files work.
- ◇ Modules & signatures can be nested inside other modules.

Modules names always begin with a capital; their contents are accessed with dot notation. Here is the general syntax:

```
(* Module declaration *)
module 'Name' : 'signature' = struct 'implementation' end

(* Signature declaration *)
module type 'Name' = sig 'sig_decls' end

(* sig_decls are a space-separated sequence of "type" or "val : 'name' : 'type'"
declarations. A "type" declaration with no "= ⋯" is an abstract type. *)
```

Without constructors `type` creates aliases, if we want to treat a type in two different ways and have the compiler ensure we don't mess-up, we produce single-constructor new types:

```
type person  = P of string;;
type address = A of string;;

let jasim : person  = P "jasim";;
let home  : address = A "the farm";;

(* Type error *)
let this : person = home;;
```

```
(* If needed, we could make coercions *)
let person_to_string : person -> string
 = function P s -> s

let address_of_string : string -> address
 = fun s -> A s
```

However, if the underlying type were the *same,* this repetition could be error-prone. Instead, we could use *generative modules*: Distinct types with the same underlying implementation. Being 'generative' is akin to the `new` keyword in Java: Each use of the `BasicImplementation` module below makes a new type.

```
module type EssentiallyString = sig
  type t
  val to_string : t -> string
  val from_string : string -> t
  val (=) : t -> t -> bool
  end
```

```
module BasicImplementaion
       : EssentiallyString = struct
  type t = string
  let to_string x = x
  let from_string x = x
  let (=) l r = String.(l = r)
end
```

Note that `BasicImplemention` is just a namespace where, e.g., `t` is an alias for `string`.

```
(* Here we get to reuse the above, thereby avoiding repetition *)
module Person  : EssentiallyString = BasicImplementaion
module Address : EssentiallyString = BasicImplementaion

let jasim : Person.t  = Person.from_string "jasim";;
let home  : Address.t = Address.from_string "the farmlands";;

(* Type error: Expected a Person.t, but have a Address.t *)
let uhoh : Person.t = home;;
```

Note that we could have placed the definitions of `EssentiallyString` and `BasicImplementaion` in-line for the `Person` and `Address` module declarations —without syntactic alterations— but that would have defeated our purpose of avoiding repetition.

Without the type annotation, we could accidentally forget to implement part of the interface & OCaml would infer a different module type. Use the `module type of` operator to see what was inferred.

```
module ErroneousImplementation = struct
  type t = string let (=) l r = String.(l = r)
end

module type S = (module type of ErroneousImplementation)
(* module type S = sig type t = string val ( = ) : 'a -> 'a -> bool end *)
```

Many *equivalent* ways to use module contents —the third is for small expressions.
 ◇ Note the dot!

```
let qasim = Person.(from_string "jasim")
in   (let open Person in qasim = jasim)
   , Person.(=) qasim jasim
   , Person.(from_string "jasim" = jasim)
   (* Rebind module name to a shorter name *)
   , let module P = Person in P.(qasim = jasim)
;;
```

While opening a module [type] affects the environment used to search for identifiers, *including* a module is a way of adding new identifiers to a module [type] proper —it is a copy-paste of constituents. For example, we could re-organise `EssentialyString` into a hierarchy:

```
module type Type = sig type t end

module type Relation = sig include Type val (=) : t -> t -> bool end
(* module type Relation = sig type t val ( = ) : t -> t -> bool end *)

module type PingPong = sig
  include Type val from_string : string -> t val to_string : t -> string end
```

This allows us to form *extensions* to modules, akin to C#: To extend, say, the `List` module with a new function `f` so that `List.f` is valid, we make and open `module List = struct let f = ··· include List`. If we wanted to *override* an existing `f`, we put its definition after the include.
The interface would be `sig val f : ··· include (module type of List) end`.

When we have multiple declarations of a type or we have type that is too abstract to be of use, expect to see: `Error: This expression has type` *t* `but an expression was expected of type M.t`; instead maybe use `M.t with type t =` *t*, where:

```
(* "Constraint Sharing": Declaration of t stays as an alias. *)
   sig type t val ··· end with type t = imp
≈  sig type t = imp val ··· end

(* "Destructive Substitution": Declaration of t is gone. *)
   sig type t val ···t··· end with type t := imp
≈  sig val ···imp··· end
```

We may now continue our `EssentiallyString` hierarchy:

```
module type IntRelation = sig include Relation with type t := int end
(* Desugars: module type IntRelation = sig val ( = ) : int -> int -> bool end *)
module type EssentiallyStr = sig include Type include Relation with type t := t end
```

Warning! The order of constructor declarations for a concrete type variant must be the same in the implementation as in the interface, likewise for record fields and function arguments.

Functors

Functors are, roughly speaking, functions from modules to modules. — There's no "modules to module types" thing, instead we return a module that contains a module type ;-) —

Functors provide the same abstraction features as functions do but also allow the implementation of one module to *vary* depending on another module. In contrast to functions, functors require explicit type annotation —otherwise how would we know what constituents the input module contains— and their arguments *must* be enclosed in parentheses.
In a sense, these are parameterised modules —as in Agda— and that is how they may be treated syntactically in OCaml:

```
(* Example functor that takes a namespace having 'x'
   and yields a namspace with 'x' incremented. *)

 module type OneInt = sig val x : int end;;
 module IncrementFunctor (M : OneInt) : OneInt = struct let x = M.x + 1 end;;

 module Three : OneInt = struct let x = 3 end;;
 module Four  : OneInt = IncrementFunctor(Three);;

 Four.x = Three.x + 1;;
```

A functor may be applied to **any** module that satisfies the functor's input interface:

```
 module Point = struct let x = 0 let y = 1 end;;
 module One = IncrementFunctor(Point);;
 One.x = Point.x + 1;;
```

First-Class Modules

Modules can contain types & values, but ordinary values can't contain modules or module types. In order to define variables whose value is a module, or a function that takes a module as an argument, OCaml provides *first-class modules* which are ordinary values that can be created from and converted back to regular modules.
A first-class module is created by packaging up a module with a signature that it satisfies by using the `module` keyword:

```
let three    : (module OneInt) = (module Three : OneInt);;
let three'   = (module Three);; (* Error: Signature couldn't be inferred *)
let three''  : (module OneInt) = (module Three);;
let three''' = (module Three : OneInt);; (* Most terse ^_^ *)

(* Lists are homogeneous, whence latters' type inferred from formers'  *)
let list_of_modules = [three; (module Three); (module Four)];;
```

Dot notation only works for records and modules, so we access contents of a first-class module by turning it into an ordinary module with `val`:

```
let x : int = let module M = (val three : OneInt) in M.x;;
let x : int = (val three : OneInt).x;; (* Error: Invalid syntax. *)
```

Warning! The parentheses for these `module, val` keywords are important!
Rule of thumb: Use the forms `(module M : T)` and `(val M : T)` always.
We can create ordinary functions which consume and create first-class modules.

```
let unpack (m : (module OneInt)) : int = let module M = (val m) in M.x;;
let pack (n : int) : (module OneInt) = (module struct let x = n end);;

3 = unpack three;;
3 = unpack (pack 3);;
pack 3 = three;; (* Woah! Equality of modules! *)

(* "Module pattern matching" *)
(* unpack' : (module OneInt) -> int *)
let unpack' (module M : OneInt) = M.x;;
```

Locally Abstract Types & '[un]bundling'

Type variables are generally implicit, but we can treat them as abstract types in a function body yet still not pass them in explicitly at use-sites.

```
(* Where did the 'a come from? *)
let wrap (x : 'a) : 'a list = [x];;

(* Let's explicitly declare it! *)
let wrap' (type a) (x : a) : a list = [x];;

(* Usage doesn't change. *)
wrap 1 = wrap' 1
```

One use is to connect the abstract types of a first-class module with other types we're working with —by using constraint sharing. *That is, we 'unbundle' the type from 'inside' the module to its 'type'.* **This is a prime reason to use first-class modules!**

Here's an example where we approximate C#'s `default` keyword:

```
module type Pointed = sig type t val point : t end;;

(* Error: The hidden type of M is exposed!
   "The type constructor M.t would escape its scope."
   "here : (M:Pointed) → M.t" *)
let default (module M : Pointed) = M.point;;

(* default : (module Pointed with type t = 'a) -> 'a *)
let default (type a) (module M : Pointed with type t = a) = M.point;;

module Nat : Pointed with type t = int = struct type t = int let point = 0 end;;
(* module Nat = struct type t = int let point = 0 end;; *)

(* 'default' works on first-class modules *)
0 = default(module Nat)
```

Conversely, this allows us to 'bundle' up data to form a module!

```
(* make_pointed : 'a -> (module Pointed with type t = 'a) *)
let make_pointed (type a) (x : a) : (module Pointed with type t = a)
  = (module struct type t = a let point = x end)
```

Metaprogramming with MetaOCaml

Quasi-quote expressions with 'brackets' `.< expr >.` *to delay their execution* and splice quoted items into such an expression with 'escape' `.~expr`. Finally, code can be 'run' with `Runcode.run expr`.

Warning! You must use the `metaocaml` command rather than the `ocaml` command.

```
open Runcode;;

(* Addition is not performed right now, but transformed into 'code' *)
let x : int code = .<1 + 2>.;; (* .<1 + 2>. *)

(* Actually execute the addition *)
let y : int = run x;;

(* Example using escape *)
let z : int code = .< let res = .~x in res + res >.;;
6 = run z;;

(* A more complicated example *)

(* Desugars using a fresh name: .<fun x_4 -> 2 * x_4>. *)
let ff = let c = 2 in .< fun x -> c * x >.;;
let six = .< .~ff 3 >.;; (* .<(fun x_14 -> 2 * x_14) 3>. *)
```

The traditional 'staging of the power function' example can be found here —as a Jupyter Notebook.

For more, see A Gentle Introduction to Multi-stage Programming

- The deprecated `!.` is replaced with `Runcode.run`.
- A Haskell rendition can be found here.

Reads

- ☒ Learn x in y minutes, where $x = OCaml$
- ☒ Try OCaml, online
- ☒ Real World OCaml
- ☐ OCaml meta tutorial —including 99 Problems
- ☐ Unix system programming in OCaml