OCaml Cheat Sheet

Functions

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

```
(* 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)

let result = fact 10

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

OCaml is a functional language: *Procedures* are functions returning the unit type.

`<fun>`

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

(* Local variables *)
;;
let x = 10 in
let y = x * 2 in
x + y
;;
let f x = x + g x and g x = x in f 2
```

```
(* Anonymouse functions *)
let sqr = fun x -> x * x

(* Only select symbols can be used as infix operators *)
let (//) x y = if x then y else false

(* (//) x y    x // y *)
let it = true // true
```

Lists

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

(* 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 ]

(* Arrays, note the dot!  *)
let xs_arr = [|1; 2; 3|]
let xs_mid = xs_arr . (1)
```

Then,

List.nth list index
List.map func list
List.filter func list
List.hd
List.rev
List.length
String.length

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)
```

Booleans

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

Strings

```
(* String catenation *)
let hw = "Hello" ^ " World"

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

References

```
(* Make a reference *)
let x = ref 1;;

(* Update the reference *)
x := 2;;

(* Use the reference *)
let y = 1 + !x
```

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)
```

User Defined Data Types

```
(* 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
```

: | (Var 666) as p -> failwith "Evil!"

```
        ^^^^^^^^^^^^^^^
Warning 26: unused variable p.
Exception: Failure "Evil!".
```

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`.

```
Characters 319-333:
```

: | (Var 666) as p -> failwith "Evil!"

```
        ^^^^^^^^^^^^^^^
Warning 26: unused variable p.
Exception: Failure "Evil!".
```

Reads

- Learn x in y minutes, where x = OCaml
- Try OCaml, online
- Real World OCaml
- Unix system programming in OCaml