O'Reilly "Real World Ocaml"

Functional PL
Strong Static Types
Interactive toploop (REPL)

```
# 3 + 4 ;;
- : int = 7
# 8 / 3 ;;
- : int = 2
# 3.5 +. 6. ;;
- : float = 9.5
# (+) ;;
- : int -> int -> int
# (+.) ;;
- : float -> float -> float
# sqrt 9. ;;
- : float = 3
# let x' = x + 1;
val x' : int = 8
```

;; is end of stmt @ top level

int_of_float
float_of_int
t1_of_t2

+ is curried

+. for floats
no auto conv like C

don't need parens for fn call

can use ' in ident

((+) 3) 4 ⇒ 7
(+) 3 4

Type inference

```
# let f x = x + 1 ;;
val f : int -> int
# f f 3 ;;  ⟶ ERROR
# f (f 3) ;;
- : int = 5
```

- fn application associates to left

Types — Ocaml (2)

3 : int 3.6 : float 'x' : char "xxx" : string () : unit

unit ← like void in C

```
# let inc x = x + 1 ;;
# let inc x = 1 + x ;;
# let inc x = (+) 1 x ;;
# let inc = (+) 1 ;;
```

(+) : int → int ⇒ β reduction

```
# (<) ;;
```

(<) : 'a → 'a → bool ≡ polymorphic

int fns : + - * / mod

float fns : +. -. *. /. **

int_of_float float_of_int

strings "abc" ^ "def" (concat)

"hello".[1] ⇒ 'e' (subscript)

String.length s

String ← module length ← fn

```
let strlen = String.length ;;
```

bool x=y x<>y x<y x>y x<=y x>=y

x<>y ≡ not(x = y)

x=y ← structural equality

x == y ← object identity

x != y

Ocaml (3)

$x == y \implies x = y$

↑ similar to Java, C++ &x == &y

↑ like Java x.equals(y), C++ x == y

~~let max x y = if x > y then x else y;;~~

~~max : 'a -> 'a -> 'a~~

```
# let abs x = if x<0 then -x else x;;
abs : int -> int
inference : (-) : int -> int
          ∴ x : int
```

f -x
f(-x)

then & else must have same type
if is an expression not a statement

```
# let max x y = if x>y then x else y
inference (>) : 'a -> 'a -> bool   (polymorphic)
```

if must be bool
x, y must be same type
However type is generic ('a)

Types: static & strict & safe
- every expr exactly one type
- no such thing as null, nullptr
- strong

Rules about type checking

1. every expr has exactly one type
2. when an expr is evaluated one of four things may happen:
   (a) evaluate to a value of same type as expr
   (b) raise an exception
   (c) may not terminate (∞ recursion)
   (d) may call exit()

Tuples, lists, options, Patterns

Tuple: comma separated item
like pair<> or tuple<> in c++

```
# let a = (3, "foo") ;;
val a : int * string = (3, "foo");
# let (x,y) = a ;;
val x : int = 3
val y : string = "foo"
# x + String.length y
  : int = 6
```

note: no parens

```
#let dist (x1,y1) (x2,y2) =
    sqrt ((x1 -. x2) **2. +. (y1 -. y2) **2.)
dist: float*float -> float*float -> float
```

- parens only for precedence
- note float ops: (-.) (**) (+.) sqrt

WARNING: (*foo*) is a comment
so is (**) but (**) is an operator
(space)

wildcards

```
#let fst (x,_) = x ;;
  fst : 'a * 'b -> 'a
#let snd (_,y) = y ;;
  snd : 'a * 'b -> 'b
```

Tuples: fixed length, heterogeneous

Lists
variable length, homogeneous
uses ; to connect

```
#let l = [1;2;3] ;;
  l : int list = [1;2;3]
```

← one token not two

```
#List.length l
-: int = 3
```

~~#List.map String.len~~

; tuples
; lists

```
# List.map ((+)1) l
  -: int list = [2;3;4]
```

((+)1) is curried partial application

note ((-)1) means (fun x -> 1 - x)
1st curried arg is left opnd.

So note:
, for tuples
; for lists

Scheme: cons car cdr '()
:: List.hd List.tl []
usually use pattern match instead of hd, tl
right assoc

[1;2;3] same as 1 :: 2 :: 3 :: []
[] is the empty list, not null or nullptr

Recursion

```
let rec fac n =
    if n <= 1 then 1 else n * fac(n-1) ;;
```

fac: int -> int

```
<= : 'a -> 'a -> bool
1 : int
∴ n : int
  fac is int -> int because 1 is int
* : int -> int -> int
- : int -> int -> int
```

fn applic higher prec than binop so need parens

But fac should be Tail-Rec

```
# let fac n =
    let rec fac' n' m' = match n' with
        | 0 -> m'
        | n' -> fac' (n'-1) (n'*m')
    in if n<0 then raise error
               else fac' n 1 ;;
```

- also intro pattern match

```
# let rec len l = match l with
    | [] -> 0
    | _::ls -> 1+ len ls ;;
# let sum l =
    let rec sum' l' m' = match l' with
        | [] -> m'
        | x::xs -> sum' xs (x + m')
    in sum' l 0 ;;
```

Options

- how to return failure or success?

Java ret ref to obj or null

C++ ret iterator to obj or c.end()

pair<K,V>

```
type 'a option = None | Some of 'a ;;
```

↑ better than an enum

constructors (must be capitalized)

```
# let divide x y =
    if y = 0 then None else Some (x/y);;
val divide : int -> int -> int option
```

(* Find value in (key * value) list *)

```
# let find eq k list = match list with
    | [] -> None
    | (k',v')::ls -> if eq k k'
                     then Some v'
                     else find eq k ls
find : ('a -> 'a -> bool) -> 'a
     -> ('a * 'b) list -> 'b option
```

ex: find (=) key list ⟹ result

pass in eq arg to make more flexible

Solve quadratic $ax^2+bx+c=0$

$$root = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

```
float -> float -> float -> float * float
let solveq a b c =
    let nb = -.b
    and sq = sqrt (b**2. -. 4. *. a *. c)
    and ta = 2. *. a
    in (nb +. sq)/.ta, (nb -. sq)/.ta
```

return both solutions as tuple

oops → √ of neg #?

Functions:
- all fns have exactly one arg and return one result
- may use tuples and currying

Records and Variants

```
type point = { x : float ; y : float }
let p = {x = 3. ; y = 4.}

let abs {x = xval; y = yval} =
    sqrt (xval ** 2. +. yval ** 2.)
```

- pattern match to extract fields

abs : point → float

```
let dist p1 p2 =
    abs {x = p1.x -. p2.x ; y = p1.y -. p2.y}
```

dist : point → point → float

-- use dot notation to select fields

-- note {x = ... ; y = ...} is a ctor

Arrays (mutable)

```
let a = [| 1;2;3;4 |];;
a : int array
a.(2) <- 4;;
```

⇒ changes value

note the .

a.(2) is subscript

a(2) ≡ a 2 is fn application

indexing starts at 0

Refs

Ocaml (10)

mutable variables

```
# let x = ref 0 ;
  val x : int ref = {contents = 0}
# !x
  - : int = 0
# x := !x + 1
  - : unit = ()
  # !x
  - : int = 1
```

! dereferences a ref.

:= is imperative store into ref.

Best to ~~:=~~

nothing built in:

```
type 'a ref = {mutable contents : 'a}
let ref x = {contents = x}
let (!) r = r.contents
let (:=) r x = r.contents ← x
```

Pattern matching & let

```
let (is, ss) = List.unzip [(1,"one"); (2,"two")]
is : int list = [1; 2]
ss : string list = ["one"; "two"]
```

how?

```
#let unzip l = match l with
    | [] → []
    | (a,b) :: ls →
        let (as, bs) = unzip ls
        in (a::as, b::bs)
```

Anonymous Functions (lambda is fun)

```
# (fun x -> x + 1)
- : int -> int
# (fun x -> x + 1) 7
- : int = 8
# List.map (fun x -> x+1) [1;2;3] ;;
- : int list = [2;3;4]
# let incf = [(fun x -> x+1); (fun x -> x+2)]
incf : (int -> int) list
# map (fun g -> g 5) incf
- : int list = [6;7]
```

List.map ((+) 1) li

what?!! (fun g -> g 5)(fun x -> x+1)

β reduction => (fun x -> x+1) 5 ⟹ 6

```
let absdiff x y = abs (x - y)
```

means:

```
let absdiff = fun x -> (fun y -> abs (x-y))
```

a function can take only one arg,

but macros expand fun appropriately

Partial application

if fn is curried can partially apply it to specialize

Prefix & infix

```
# max 3 4
: int = 4
# 3 + 4
: int = 7
# (+) 3 4
: int 7
# (+) 3
: int -> int
# List.map ((+)3) [1;2;3]
: int list = [4;5;6]
```

(-)3 ≡ (fun x -> 3-x)

⟹ 3 is LEFT arg

operator chars. any of
! $ % & + * - . / : <=> ? @ ^ | ~

except: *

precedence determined by 1st char

```
# let (+!) (x1,y1) (x2,y2) = (x1+x2, y1+y2)
val (+!) : int*int -> int*int -> int*int
# (3,2) +! (-2,4)
: int*int = (1,6)
```

But Beware: (* starts a comment

Precedence & associativity

prefix	!... ?... ~...
nonassoc	. .(.[
left	fn applic, ctor, assert, lazy
prefix	- -.
right	**... lsl lsr asr
left	*... /... %... mod land lor lxor
left	+... -...
right	::
right	@... ^...
left	=... <... >... |... &... $...
right	&&
right	||
N/A	,
right	<- :=
N/A	if
right	;

underlined are reserved words
... means any oper beginning with

```
let (|>) x f = f x   /// like Unix pipe
```

ex: `split ':' path |> dedup |> iter print`

means:
```
let s = split ':' path in
    let t = dedups in
    iter print t ;;
```

Lists & Patterns

`[1;2;3]` ⟹ `1 :: 2 :: 3 :: []`

right associative

```
let rec sum l = match l with
    | [] -> 0
    | x :: xs -> x + sum xs
```

Tail-rec:

```
let sum l =
    let rec sum' l' m' = match l' with
        | [] -> m'
        | x::xs -> sum' xs (x + m')
    in sum' l 0
```

patterns match data structures and constants but not arbitrary conditions

case matching looks sequential
but optimized for direct jump
like switch in C++/Java

```
# let foo x = match x with
    | 0 -> 1
    | 1 -> 2
    | 2 -> 5
    | 6 -> 10
    | _ -> x*10 ;;
```

. _ is wildcard pattern match

Programs

no main function
just exec all global stmts (exprs)

ocamlc - compile to bytecode
ocamlopt - compile to native code

Multiple source files
- collection of modules
  - .mli - interface
  - .ml - implementation

ML ancestor of Caml (INRIA) ⇒ Ocaml
(Robin Milner, author)

ocamlc -i foo.ml
builds interface file

(INRIA = Institute National de Recherch en Informatique et d'Automatique

Opening

To avoid mentioning modules

```
open Printf;;
open List;;
open String;;
```

} like import in Java

or:

```
let strlen = String.length
let strcmp = String.compare
```

Sys.argv is an array
Sys.argv.(i) is a sys arg
(note the dot)

ex: (Filename.basename Sys.argv.(0))

Including

```
include Lexing
include Foomodule
```

but: cyclic dependencies not permitted

Variants

-- like enum

```
type color = Red | Green | Blue
```

(color: not capitalized; Red, Green, Blue: Must be capitalized)

```
let string_of_color = function
  | Red -> "red"
  | Green -> "green"
  |Blue -> "blue" ;;
```

Recursive data structures

AST (abstract syntax tree)

```
type ast = Add of ast*ast
         | Mul of ast*ast
         | Number of float
         | Ident of string
         | Assign of ast*ast ;;
```

a = b*c + d*e

~~Assign "a" (Add (Mul~~

```
let e = Assign ("a",
                Add (Mul (Ident "b",
                          Ident "c"))
                    (Mul (Ident "d",
                          Ident "e"))) ;;
```

Exceptions

```
# 3/0 ;;
  Exception: division_by_zero
```

declaring

```
# exception Foobar
```

must be cap (ctor)

```
# exception Bar of string
```

∴ raise Foobar
raise (Bar "what is")

Helpers

```
# let failwith s = raise (Failure s)
```

assert (bool cond)

```
try
  expr
with
    | exn1 -> expr1
    | exn2 -> expr2
```

} like try/catch in Java C++

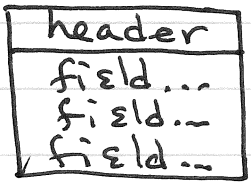
Printf


```
printf "%i is a %s\n" 3 "foo" ;;
```

but strings are type checked

```
let fmt = ('a,'b) format = "%i %s";;
printf fmt i s;;
```

Memory objects



— contains size

field: if pointer LSB = 0
if int LSB = 1
∴ uses 31 bit ints on 32 bit sys
63 bit ints on 64 bit sys

so if ptrs are 64 bits
ints are -2^{62} to $2^{62}-1$

Garbage collection

— of course