OCaml

Yuxin Deng

yxdeng@sei.ecnu.edu.cn

Ocaml and UTop

1. Install ocaml via opam
2. Install utop via opam

   https://opam.ocaml.org/blog/about-utop/

   opam install utop

   eval `opam config env` # may not be needed

   opam init

   utop

Hello World!

Create a ”hello.ml”file:

```
print_endline ”Hello World!”
```

Run it with the interpreter:

```
$ ocaml hello.ml
```

Run it with the bytecode interpreter:

```
$ ocamlc –o hello hello.ml
$ ocamlrun hello
```

On most systems, the bytecode can be run directly:

```
$ ocamlc –o hello hello.ml
$ ./hello
```

Interpret and Compile

Compile a native executable and run:

```
$ ocamlopt -o hello hello.ml
$ ./hello
```

The interactive Read-Eval-Print Loop

```
$ ocaml
# print_endline  “Hello World!”  ;;

# #use  “hello.ml”  ;;

# #quit;;
$
```

Running OCaml code

```
$ ocaml
        OCaml version 4.09.0
# 1+1;;
- : int = 2

# ”  Hello”  ^ ”  World!”  ;;
```

Defining a function

```
# (* define a function *)
  let average a b =
    (a +. b) /. 2.0;;
val average : float -> float -> float = <fun>

(* OCaml defines + as the integer addition function. To add floats, use +. (note the trailing
period). Similarly, use -., *., /. for other float operations. *)

# average 3. 4.;;
- : float = 3.5

# let plus = fun x y -> x + y ;;
val plus : int -> int -> int = <fun>
# plus 2 3;;
```

Defining a function

```
# let a = 3 in a + 5 ;;
- : int = 8

# (fun a -> a + 5) 3 ;;
- : int = 8
```

Semantically equivalent; ‘let’ is easier to read.

Polymorphism

```
# let f x = x ;;
val f : 'a -> 'a = <fun>
# f 3 ;;
- : int = 3
# f true ;;
- : bool = true
# f print_int ;;
- : int -> unit = <fun>
# f print_int 1 ;;
1- : unit = ()
```

Polymorphism

Ocaml always infers the most general type.

```
# let compose f g = fun x -> f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Basic types

```
OCaml type  Range

int         31-bit signed int (roughly +/- 1 billion) on 32-bit
            processors, or 63-bit signed int on 64-bit processors
float       IEEE double-precision floating point, equivalent to C's double
bool        A boolean, written either true or false
char        An 8-bit character
string      A string
unit        Written as ()
```

Unit type

Expressions with no meaningful value (assignment, loop, ⋯) have type unit.
This type has a single value, written ().
It is the type given to the else branch when it is omitted.

Correct:

```
if !x > 0 then x := 0
```

Incorrect:

```
2 + (if ! X > 0 then 1)
```

Implicit vs. explicit casts

```
# 1 + 2.5 ;;
Error: This expression has type float but an expression was expected of type
         int

# 1 +. 2.5 ;;
Error: This expression has type int but an expression was expected of type
         float

# (float_of_int 1) +. 2.5 ;;
- : float = 3.5
```

Lists

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

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

Array

```
# let a = [| 1; 3; 5 |] ;;
val a : int array = [|1; 3; 5|]

# a.(0) ;;
- : int = 1

# a.(2) <- 7 ;;
- : unit = ()

# a ;;
- : int array = [|1; 3; 7|]

# let a = Array.make 5 "a" ;;
val a : string array = [|"a"; "a"; "a"; "a"; "a"|]
```

Structures or records

```
# type pair_of_ints = { a: int; b : int };;
type pair_of_ints = { a : int; b : int; }

# {a=3; b=5};;
-   : pair_of_ints = {a = 3; b = 5}


# type complex = {re : float; im : float} ;;
type complex = { re : float; im : float; }

# let x = { re = 1.0; im = -1.0} ;;
val x : complex = {re = 1.; im = -1.}

# x.im ;;
- : float = -1.
```

Mutable fields

```
# type person = {name : string; mutable age : int} ;;
type person = { name : string; mutable age : int; }

# let p = { name = "John"; age = 20};;
val p : person = {name = "John"; age = 20}

# p.age <- p.age + 1;;
- : unit = ()

# p.age;;
- : int = 21
```

Variants

```
# type foo =
    | Nothing
    | Int of int
    | Pair of int * int
    | String of string;;
type foo = Nothing | Int of int | Pair of int * int | String of string

# Pair (3, 4);;
- : foo = Pair (3, 4)

# type sign = Positive | Zero | Negative ;;
type sign = Positive | Zero | Negative

# Positive ;;
- : sign = Positive
```

Recursive variants

```
# type binary_tree =
    | Leaf of int
    | Tree of binary_tree * binary_tree;;
type binary_tree = Leaf of int | Tree of binary_tree * binary_tree

# Leaf 3;;
- : binary_tree = Leaf 3

# Tree (Tree (Leaf 3, Leaf 4), Leaf 5);;
- : binary_tree = Tree (Tree (Leaf 3, Leaf 4), Leaf 5)
```

Parameterized variants

```
# type 'a binary_tree =
    | Leaf of 'a
    | Tree of 'a binary_tree * 'a binary_tree;;
type 'a binary_tree = Leaf of 'a | Tree of 'a binary_tree * 'a binary_tree

# Tree (Leaf "hello", Tree (Leaf "John", Leaf "Smith"));;
- : string binary_tree =
Tree (Leaf "hello", Tree (Leaf "John", Leaf "Smith"))

# Leaf 3.1;;
-  : float binary_tree = Leaf 3.1
```

Parameterized variants

```
# type 'a list =
    | Nil
    | Cons of 'a * 'a list;;
type 'a list = Nil | Cons of 'a * 'a list

# Nil;;
- : 'a list = Nil

# Cons (1, Nil);;
- : int list = Cons (1, Nil)

# Cons (1.1, Cons(2.1, Nil));;
- : float list = Cons (1.1, Cons (2.1, Nil))
```

Local “variables” (really local expressions)

```
# let average a b =
    let sum = a +. b in
    sum /. 2.0;;
val average : float -> float -> float = <fun>

# let f a b =
    (a +. b) +. (a +. b) ** 2. ;;
val f : float -> float -> float = <fun>

# let f a b =
    let x = a +. b in
    x +. x ** 2. ;;
val f : float -> float -> float = <fun>
```

Global “variables” (really global expressions)

```
let html =
  let content = read_whole_file file in
  GHtml.html_from_string content
  ;;

let menu_bold () =
  match bold_button#active with
  | true -> html#set_font_style ~enable:[`BOLD] ()
  | false -> html#set_font_style ~disable:[`BOLD] ()
  ;;

let main () =
  (* code omitted *)
  factory#add_item "Cut" ~key:_X ~callback: html#cut
  ;;
```

References (real variables)

```
# ref 0;;
-  : int ref = {contents = 0}

# let my_ref = ref 0;;
val my_ref : int ref = {contents = 0}

# my_ref := 100;;
- : unit = ()

# !my_ref;;
- : int = 100
```

References (real variables)

A reference = a record of predefined type

type 'a ref = { mutable contents : 'a }

ref, ! and := are syntactic sugar
Only arrays and mutable fields can be mutated

Tuples

Usual notation

```
# (1,2,3);;
- : int * int * int = (1, 2, 3)

# let v = (0, false, "window", 'a') ;;
val v : int * bool * string * char = (0, false, "window", 'a’   )
```

Access to components

```
# let (a, b, c, d) = v ;;
val a : int = 0
val b : bool = false
val c : string = "window"
val d : char = 'a’

# print_string c;;
window- : unit = ()
```

Tuples

Useful to return several values

```
# let rec division n m =
  if n < m then (0, n)
  else let (q, r) = division (n-m) m in
         (q + 1, r) ;;
val division : int -> int -> int * int = <fun>
```

Function taking a tuple as argument

```
# let f (x,y) =  x + y ;;
val f : int * int -> int = <fun>

# f (1,2);;
- : int = 3
```

In and Out

```
# print_int 100;;
100- : unit = ()

# let print_pair (x, y) =
  print_int x; print_newline (); print_int y; print_newline () ;;
val print_pair : int * int -> unit = <fun>

# print_pair (1,2);;
1
2
- : unit = ()

# let a = read_int ()
        in print_int a;;

# let name = read_line ()
      in print_string name ;;
```

No null value

In Ocaml, there is no null value.
Any value is necessarily initialized.

An expression of type τ whose evaluation terminates necessarily has a legal value of type τ.

This is known as strong typing.
No such thing as NullPointerException

Nested functions

```
# let read_whole_channel chan =
    let buf = Buffer.create 4096 in
    let rec loop () =
      let newline = input_line chan in
      Buffer.add_string buf newline;
      Buffer.add_char buf '\n';
      loop ()
    in
    try
      loop ()
    with
      End_of_file -> Buffer.contents buf;;
val read_whole_channel : in_channel -> string = <fun>
```

Pattern matching

```
# type expr =
    | Plus of expr * expr        (* means a + b *)
    | Minus of expr * expr       (* means a - b *)
    | Times of expr * expr       (* means a * b *)
    | Divide of expr * expr      (* means a / b *)
    | Value of string          (* "x", "y", "n", etc. *);;
type expr =
    Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr
  | Divide of expr * expr
  | Value of string
```

```
# let rec to_string e =
    match e with
    | Plus (left, right) ->
       "(" ^ to_string left ^ " + " ^ to_string right ^ ")"
    | Minus (left, right) ->
       "(" ^ to_string left ^ " - " ^ to_string right ^ ")"
    | Times (left, right) ->
       "(" ^ to_string left ^ " * " ^ to_string right ^ ")"
    | Divide (left, right) ->
       "(" ^ to_string left ^ " / " ^ to_string right ^ ")"
    | Value v -> v;;
val to_string : expr -> string = <fun>

# let print_expr e =
    print_endline (to_string e);;
val print_expr : expr -> unit = <fun>

# print_expr (Times (Value "n", Plus (Value "x", Value "y")));;
(n * (x + y))
- : unit = ()
```

Wildcards

Underscore (_) is a wildcard that will match anything, useful as a default or when you just don’ t care.

```
#  let xor p = match p
    with (true, false) | (false, true) -> true
    | _ -> false;;
val xor : bool * bool -> bool = <fun>

# xor (true, true);;
- : bool = false
```

If statements, loops

```
# let max a b =
    if a > b then a else b;;
val max : 'a -> 'a -> 'a = <fun>

# max 2 3;;
- : int = 3

# max 2.1 5.4;;
- : float = 5.4

# max "a" "b";;
- : string = "b"
```

```
# let sum = ref 0 in
  for i = 1 to 10 do
    sum := !sum + i
  done      ;
  print_int !sum;;
55- : unit = ()
```

```
# let flag = ref false in
  while not !flag do
    print_string "Terminate? (y/n)";
    let str = read_line () in
    if str.[0] = 'y' then
      flag := true
  done;;
Terminate? (y/n)n
Terminate? (y/n)abc
Terminate? (y/n)y
- : unit = ()
```

While loops

```
# let array_mem x a =
  let len = Array.length a in
  let flag = ref false in
  let i = ref 0 in
     while !flag = false && !i < len do
       if a.(!i) = x then
          flag := true;
       i := !i + 1
     done;
     !flag ;;
val array_mem : 'a -> 'a array -> bool = <fun>

# array_mem 1 [| 3; 1; 6; 7 |];;
- : bool = true
```

```
# let array_mem' x a =
  let flag = ref false in
    for i = 0 to Array.length a - 1 do
      if a.(i) = x then
        flag := true
    done;
    !flag ;;
val array_mem' : 'a -> 'a array -> bool = <fun>

# array_mem' 1 [| 3; 5; 1; 7 |];;
- : bool = true

# array_mem 7 [| 3; 5; 1; 6|];;
- : bool = false
```

Recursive functions

```
# let rec range a b =
    if a > b then []
    else a :: range (a+1) b;;
val range : int -> int -> int list = <fun>

# range 1 5;;
- : int list = [1; 2; 3; 4; 5]
```

```
# let rec range2 a b accum =
    if b < a then accum
    else range2 a (b-1) (b :: accum);;
val range2 : int -> int -> int list -> int list = <fun>

# let range a b =
    range2 a b [];;
val range : int -> int -> int list = <fun>
```

Recursive functions

```
# let rec fact n =
    if n = 0 then 1
    else n * fact (n-1);;
val fact : int -> int = <fun>

# fact 3;;
- : int = 6

# fact 5;;
- : int = 120
```

```
# let rec list_max list =
    match list with
    | [] -> failwith "list_max called on empty list"
    | [x] -> x
    | x :: list' -> max x (list_max list');;
val list_max : 'a list -> 'a = <fun>

# list_max [1; 3; 5; 4; 2];;
- : int = 5
```

Mutual recursion

```
# let rec fact n =
    if n = 0 then 1
    else n * fact1 n
and fact1 = fact (n-1);;
val fact : int -> int = <fun>
val fact1 : int -> int = <fun>

# fact 5;;
- : int = 120
```

Imperative feature

```
# let rec fact n =
    if n = 0 then 1
    else n * fact1 n
and fact1 = fact (n-1);;
val fact : int -> int = <fun>
val fact1 : int -> int = <fun>

# fact 5;;
- : int = 120
```

vs

```
# let fact n =
    let result = ref 1 in
    for i = 2 to n do
      result := i * !result
    done;
  !result ;;
val fact : int -> int = <fun>

# fact 5;;
- : int = 120
```

Ackermann functions

$$A(m,n) = \begin{cases} n + 1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

where m and n are non-negative integers

Ackermann functions

```
# let rec ack m n =
if m = 0 then n + 1
else if n = 0 then ack (m-1) 1
     else if m > 0 && n > 0 then ack (m-1) (ack m (n-1))
          else failwith    “Negative parameters”
;;
val ack : int -> int -> int = <fun>
```

```
# #use "test.ml";;
val ack : int -> int -> int = <fun>
# ack 3 3;;
- : int = 61
```

Tree exercises

```
# type 'a bt =
  Lf
  | Br of 'a * 'a bt * 'a bt ;;

# let rec size tr =
  match tr with
   | Lf -> 0
   | Br (_, l, r) -> 1 + size l + size r ;;
val size : 'a tree -> int = <fun>

# let rec total tr =
  match tr with
   | Lf -> 0
   | Br (x, l, r) -> x + total l + total r ;;
val total : int tree -> int = <fun>
```

```
# let max x y =
  if x > y then x else y


let rec maxdepth tr =
  match tr with
  | Lf -> 0
  | Br (_, l, r) -> 1 + max (maxdepth l)
(maxdepth r);;
val max : 'a -> 'a -> 'a = <fun>
val maxdepth : 'a tree -> int = <fun>

# let rec list_of_tree tr =
  match tr with
  | Lf -> []
  | Br (x, l, r) -> list_of_tree l @ [x] @
list_of_tree r ;;
val list_of_tree : 'a tree -> 'a list = <fun>
```

Higher-order function

```
# let rec map f l =
    match l with
     [] -> []
    | h :: t ->  f h :: map f t ;;

let halve x = x / 2 ;;

# map halve [10; 20; 30];;
- : int list = [5; 10; 15]
```

Higher-order function

```
# let rec power f n =
    if n = 0 then fun x -> x
    else compose f (power f (n-1))
and compose f g = fun x -> f (g x) ;;
val power : ('a -> 'a) -> int -> 'a -> 'a = <fun>
val compose : ('a -> 'a) -> ('a -> 'a) -> 'a -> 'a = <fun>

# let derivative dx f = fun x -> (f (x +. dx) -. f x) /. dx ;;
val derivative : float -> (float -> float) -> float -> float = <fun>

# let sin''' = power (derivative 1e-5) 3 sin;;
val sin''' : float -> float = <fun>
```

Higher-order function

```
# let integral f =
    let n = 100 in
    let s = ref 0.0 in
    for i = 0 to n-1 do
      let x = float i /. float n in
        s := !s +. f x
    done;
    !s /. float n ;;

val integral : (float -> float) -> float = <fun>

# integral sin;;
- : float = 0.45548650838731830 1

# integral (fun x -> x *. x) ;;
- : float = 0.32835000000000031
```

Insertion sort

```
# let rec sort = function
    | [] -> []
    | x :: l -> insert x (sort l)
 and insert a = function
        | [] -> [a]
        | x :: l -> if a < x then a :: x :: l
                    else x :: insert a l ;;
val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>

# sort [2; 1; 3; 0];;
- : int list = [0; 1; 2; 3]

# sort ["how"; "are"; "you"] ;;
- : string list = ["are"; "how"; "you"]
```

Sort is defined by two recursive functions.

The type of the list elements are unspecified. It’ s represented by a type variable ‘a.

Quicksort

```
# let rec quicksort l =
      match l with
      | [] -> []
      | [x] -> [x]
      | h::tl ->
        let rec partition list =
          match rest with
          | [] -> [], []
          | a::list'    ->
              let (left, right) = partition list'    in
               if a < h then. (a :: left, right)
              else  (left, a :: right)
             in  let (l, r) = partition tl in
           (quicksort l) @ (h :: quicksort r);;
val quicksort : 'a list -> 'a list = <fun>

# quicksort [2; 5; 1; 7; 3; 9; 3; 0; 10];;
- : int list = [0; 1; 2; 3; 3; 5; 7; 9; 10]
```

Exceptions

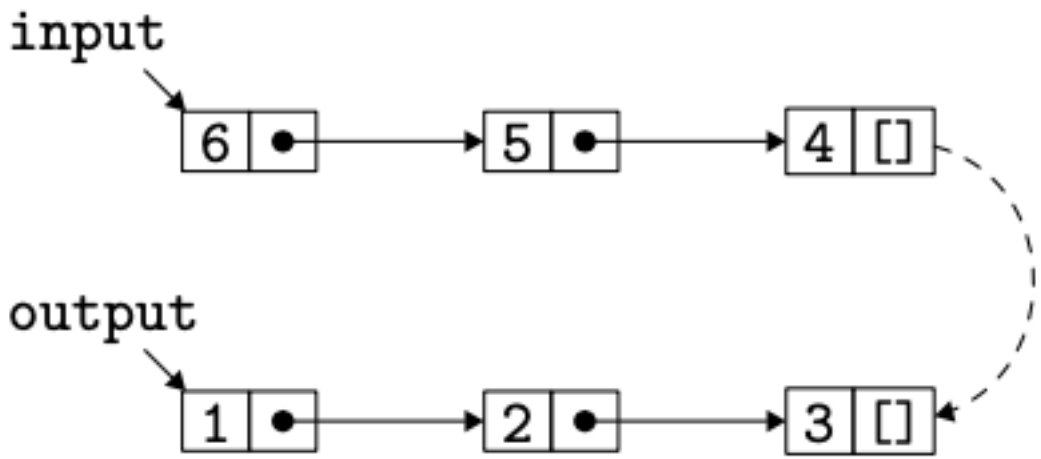
```
# 1 / 0;;
Exception: Division_by_zero.

# try
 1 / 0
 with Division_by_zero -> 13;;
- : int = 13

# exception My_exception ;;
exception My_exception

# try
  if true then
    raise My_exception
  else 0
  with My_exception -> 13;;
- : int = 13
```

Exceptions

```
# exception Exception1 of string;;
# exception Exception2 of int * string;;

# let except b =
 try
   if b then
     raise (Exception1 "aaa")
   else
     raise (Exception2 (13, "    bbb"))
 with Exception1 s -> "Exception1: "^s
     |Exception2 (n, s) -> "Exception2 "^string_of_int n ^ s ;;
val except : bool -> string = <fun>

# except true;;
- : string = "Exception1: aaa"

# except false;;
- : string = "Exception2 13 bbb"
```

Queues

Idea: a queue is a pair of lists, one for insertion, and one for extraction.



This stands for the queue -> 6, 5, 4, 3, 2, 1 ->

Queues

```
# let create () = [], [];;
val create : unit -> 'a list * 'b list = <fun>

# let push x (i, o) = (x :: i, o) ;;
val push : 'a -> 'a list * 'b -> 'a list * 'b = <fun>

# let pop q =
  let (i, o) = q in
    match o with
     | x :: o' -> x, (i, o')
     | [] -> match List.rev i with
               | x :: i' -> x, ([], i')
               | [] -> raise Empty ;;
val pop : 'a list * 'a list -> 'a * ('a list * 'a list) = <fun>
```

```
# push 2 ([1],[]);;
- : int list * 'a list = ([2; 1], [])
# pop ([2; 1], []);;
- : int * (int list * int list) = (1, ([], [2]))
```