

## 1 Examen 2020

1.1 (1 punto) Indique el efecto de la compilación y ejecución de las siguientes frases, como lo indicaría el compilador interactivo de Ocaml. Distinga claramente expresiones de definiciones.

```
let f = List.fold_left(fun x y -> 2*x +y) 0;;
(*-----results-----*)
val f : int list -> int = <fun>

f [1;0;1;1],f[1;1;1;1];;
(*-----results-----*)
- : int * int = (11, 15)

let rec base b n =
  let q = n/b in
  if q = 0 then [n]
  else n mod b::base b q;;
(*-----results-----*)
val base : int -> int -> int list = <fun>

[base 2 1; base 2 16; base 10 2021];
(*-----results-----*)
- : int list list = [[1]; [0; 0; 0; 0; 1]; [1; 2; 0; 2]]
```

1.2 Considere la función f definida en Ocaml a continuación:

```
let f l =
  if l = [] then [] else
    let l = ref l and r = ref [] in
    while List.tl !l <> [] do
      if List.hd !l < List.hd t then
        r:= List.hd t:: !r
      else ();
      l:=t
    done;
    List.rev !r;;
```

*falta código no se puede resolver*

1.2.1 (0.25 puntos) Indique el tipo de la función f.

*supuestamente*

```
val f: 'a list -> 'a list = <fun>
```

1.2.2 (0.25 puntos) Indique el tipo y valor de la siguiente expresión en Ocaml

```
f [1;2;3;0;9];;
```

*el tipo de dato será 'a list, el valor no lo podemos saber al no estar completa la función*

1.2.3 (0.75 puntos) Realice una implementación funcional (recursiva, sin bucles ni referencias), lo más sencilla que pueda, para la función f.

```
(*
*
*
*
*
*
*
*
*
*)
```

## 2 Examen 2019

2.1 (4 puntos) Indique el efecto de la compilación y ejecución de las siguientes frases, como lo indicaría el compilador interactivo de Ocaml. Distinta claramente expresiones de definiciones.

```
let x,y = 2+1,0;;
(*-----results-----*)
val x : int = 3
val y : int = 0
```

```
(function x -> function y -> 2*y) y x;;
(*-----results-----*)
- : int = 6
```

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

```
let g f x = f (f x) in g f 5;;
(*-----results-----*)
- : int = 11
```

```
let h = fun x y -> y::x;;
(*-----results-----*)
val h : 'a list -> 'a -> 'a list = <fun>
```

```
h ['h'];;
(*-----results-----*)
- : char -> char list = <fun>
```

```
h [] [0];;
(*-----results-----*)
- : int list list = [[0]]
```

```
let x,y = y,x;;
(*-----results-----*)
val x : int = 0
val y : int = 3
```

```
let v = ref x;;
(*-----results-----*)
val v : int ref = {contents = 0}
```

```
v+1;;
(*-----results-----*)
Line 1, characters 0-1:
1 | v+1;;;
   ^
```

Error: This expression has type `int ref`  
but an expression was expected of type `int`

```
let w=v;;
(*-----results-----*)
val w : int ref = {contents = 0}
```

```
w:=!w+1;!v,!w;;
(*-----results-----*)
- : int * int = (1, 1)
```

## 2.2 Considere la siguiente definición en Ocaml.

```
let rec trivide = function
  [] -> [], [], []
| h::t -> let t1, t2, t3 = trivide t in
          h::t3,t1,t2;;
```

### 2.2.1 Indique el tipo de la función trivide.

```
val trivide : 'a list -> 'a list * 'a list * 'a list = <fun>
```

### 2.2.2 Si la implementación dada es recursiva terminal indíquelo razonadamente y si no lo es realice una implementación alternativa que sí lo sea.

```
let trivide l =
  let rec aux s1 s2 s3 = function
    [] -> List.rev s1, List.rev s2, List.rev s3
  | h1::h2::h3::t -> aux (h1::s1) (h2::s2) (h3::s3) t
  | h1::h2::t -> aux (h1::s1) (h2::s2) s3 t
  | h1::t -> aux (h1::s1) s2 s3 t
  in aux [] [] [] l;;
```

## 2.3 (1 punto) Considere la siguiente definición en Ocaml.

```
type ('a,'b) tree= S of 'b|T of 'a* ('a,'b) tree*('a,'b)tree;;
```

Considere también la siguiente definición en Ocaml:

```
let rec eval = function
  S x -> x
| T (op,t1,t2) -> op (eval t1) (eval t2);;
```

### 2.3.1 Indique el tipo de la función eval

```
val eval : ('a -> 'a -> 'a, 'a) tree -> 'a = <fun>
```

### 2.3.2 Considere las siguientes definiciones

Indique el tipo de e y el valor de x.

```
let e = T(( * ), T((+), S 2, S 3), S 5);;
let x = eval e;;
(*-----results-----*)
val e : (int -> int -> int, int) tree = T (<fun>, T (<fun>, S 2, S 3), S 5)
val x : int = 25
```

## 3 PP Julio 2019

### 3.1 (4 puntos) INdique el efecto de la compilación y ejecución de las siguientes frases, como lo indicaría el compilador interactivo de Ocaml. Distinga claramente expresiones de definiciones.

```
List.map (fun x-> x,x) ['a';'e';'i'];;
(*-----results-----*)
- : (char * char) list = [('a', 'a'); ('e', 'e'); ('i', 'i')]
```

```
let x= 1 in
  for i=1 to 3 do
    let x = 2*x in print_int x
  done;
  print_int x;;
(*-----results-----*)
2221- : unit = ()
```

```

let rec fold op e = function
  [] -> e
  | h::t -> fold op (op e h) t;;
(*-----results-----*)
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

```

```

let f = fold (fun n _ -> n `1) 0 in f ['a'; 'e'; 'i'];;
(*-----results-----*)
está mal escrita

```

```

let f = fold (+) 0 in f[1;2;3];;
(*-----results-----*)
- : int = 6

```

```

let rec repeat n f x =
  if n > 0 then repeat (n+1) f (f x)
  else x;;
(*-----results-----*)
val repeat : int -> ('a -> 'a) -> 'a -> 'a = <fun>

```

```

let push f(h::t) = f h::h::t;;
let succ = (+) 1 in repeat 3 (push succ) [0];;
(*-----results-----*)
bucle infinito

```

### 3.2 Considere la función criba definida en Ocaml a continuación.

```

let rec criba = function
  x::[] -> [x]
  | x::_:t -> x::criba t
  | _ -> [];;

```

#### 3.2.1 (0.5 puntos) Indique el tipo de la función criba.

```

val criba : 'a list -> 'a list = <fun>

```

#### 3.2.2 (0.5 puntos) Indique el tipo y valor de la siguiente expresión Ocaml.

```

criba ['a'; 'e'; 'i'; 'o'; 'u'];;
(*-----results-----*)
- : char list = ['a'; 'i'; 'u']

```

#### 3.2.3 (1.5 puntos) Si la implementación dada para criba es recursiva terminal indíquelo razonablemente y si no lo es realice una implementación alternativa que sí lo sea.

```

let criba l =
  let rec aux sol = function
    [] -> sol
    | h1::[] -> h1::sol
    | h1::h2::t -> aux (h1::sol) t
  in List.rev (aux [] l);;

```

## 4 PP 2ª oportunidad 2018

#### 4.1 (4 puntos) Indique el efecto de la compilación y ejecución de las siguientes frases, como lo indicaría el compilador interactivo de Ocaml. Distinga claramente expresiones de definiciones.

```

let x y = y,y;;
(*-----results-----*)
val x : 'a -> 'a * 'a = <fun>

```

```

    let x = (+) 2 in List.map x [1;2;3;101];;
  (*-----results-----*)
  - : int list = [3; 4; 5; 103]

  [x 10];;
  (*-----results-----*)
  - : (int * int) list = [(10, 10)]

  let appto x f = f x;;
  (*-----results-----*)
  val appto : 'a -> ('a -> 'b) -> 'b = <fun>

  List.map (appto 101) [abs;pred;succ;(+) 2];;
  (*-----results-----*)
  - : int list = [101; 100; 102; 103]

  let tail l = try List.tl l with _ -> [];;
  (*-----results-----*)
  val tail : 'a list -> 'a list = <fun>

  let l = ['a';'e'] in let l = tail l in let l2 = tail l in [l, l2, tail l2];;
  (*-----results-----*)
  - : (char list * char list * char list) list = [[['e'], [], []]]

  let x, y = let x = 5. in let y = x /. 2. in 2. *. y, 2. *. x;;
  (*-----results-----*)
  val x : float = 5.
  val y : float = 10.

```

## 4.2 Considere la siguiente definición en Ocaml.

```

let rec comp l x = match l with
  [] -> x
  | h::t -> h (comp t x);;

```

### 4.2.1 Indique el tipo de la función comp.

```

val comp : ('a -> 'a) list -> 'a -> 'a = <fun>

```

### 4.2.2 Si la implementación dada es recursiva terminal indíquelo razonadamente y si no lo es realice una implementación alternativa que sí lo sea.

```

let comp l x =
  let rec aux r = function
    [] -> r
    | h::t -> aux (h r) t
  in aux x (List.rev l);;

```

## 4.3 (1.5 puntos) Considera la siguiente definición imperativa en Ocaml.

```

let lmax l=
  let m = ref (List.hd l) in
  let l = ref (List.tl l) in
  while !l <> [] do
    m := max !m (List.hd !l);
    l := List.tl (!l)
  done;
  !m;;

```

#### 4.3.1 Indique el tipo de la función lmax

```
val lmax : 'a list -> 'a = <fun>
```

#### 4.3.2 Implemente en Ocaml una versión funcional que sea recursiva tearminal de la función lmax.

```
(*  
**  
**  
**  
**  
**  
**  
**  
**  
*)
```

#### 4.4 (1 punto) Utilizaremos la siguiente definición para representar, con valores tipo 'a tree, árboles binarios con valores de tipo 'a asociados a los nodos.

```
type 'a tree = Empty | Tree of 'a tree * 'a * 'a tree;
```

Defina en Ocaml, del modo más sencillo posible, una función mirror: 'a tree \* 'a tree -> bool, que indique para cada dos árboles si uno de ellos es la imagen especular del otro.

```
(*  
**  
**  
**  
**  
**  
**  
**  
**  
*)
```

## 5 PP 1ª oportunidad 2018.

#### 5.1 (4 puntos). Indique el efecto de la compilación y ejecución de las siguientes frases, como lo indicaría el compilador interactivo de Ocaml. Distinga claramente expresiones de definiciones.

```
2 * min_int;;  
(*-----results-----*)  
- : int = 0
```

```
let x::y = ['a'; 'e'; 'i'; 'o'; 'u'];;  
(*-----results-----*)  
val x : char = 'a'  
val y : char list = ['e'; 'i'; 'o'; 'u']
```

```
List. tl y;;  
(*-----results-----*)  
- : char list = ['i'; 'o'; 'u']
```

```
x::y;  
(*-----results-----*)  
- : char list = ['a'; 'e'; 'i'; 'o'; 'u']
```

```
let f x y = 2* (x*y);;  
(*-----results-----*)  
val f : int -> int -> int = <fun>
```

sospecho que está mal copiada esta función

```
f 10;;
(*-----results-----*)
- : int -> int = <fun>

let f = f 0;;
(*-----results-----*)
val f : int -> int = <fun>

let l = [0;1;2;3;4;5] in List.map f l;;
(*-----results-----*)
- : int list = [0; 0; 0; 0; 0; 0]
```

## 5.2 (1.5 puntos) Defina en Ocaml, del modo más sencillo posible lo siguiente:

La función lmax: 'a list -> 'a list -> 'a list del modo que, al aplicarla a dos listas con el mismo número de elementos devuelva una lista en la que cada elemento sea mayor (según el orden ( $\leq$ )) de los elementos que ocupan esa misma posición en esas dos listas. Si las listas no tienen la misma longitud debe provocar la excepción Invalid\_argument "lmax". Así, por ejemplo, en el compilador interactivo, podríamos comprobar...

```
lmax [2;0;3] [1;1;2];;
(*-----results-----*)
- : int list = [2;1;3]

lmax ["uno"; "dos"; "tres"; "cuatro"]
     ["dos"; "tres"; "cuatro"; "cinco"];
(*-----results-----*)
- : string list = ["uno"; "tres"; "tres"; "cuatro"]

lmax [1;2;3] [1;2;3;4];;
(*-----results-----*)
Exception: Invalid_argument "lmax".
```

code:

```
let rec lmax l1 l2 = match (l1,l2) with
| [], [] -> []
| a::l3, b::l4 ->
    if List.length l1 <> List.length l2 then raise (Invalid_argument "lmax")
    else if a <= b then b::lmax l3 l4
    else a::lmax l3 l4
| _ -> [];;
```

## 5.3 Ejercicios árboles:

### 5.3.1 (1 punto) Defina en Ocaml una función leafs: treeSk -> int que, para cada estructura del árbol,

devuelva el número de hojas que contiene (Así, por ejemplo, tendríamos leafs t1=1 y leafs t5=5);

```
(*
*
*
*
*
*
*
*)
```

**5.3.2 (0.75 puntos)** Defina en Ocaml una función `mirror: treeSk -> TreeSk` que, para cada estructura

de árbol, devuelva su imagen especular.

```
(*  
*  
*  
*  
*  
*  
*  
*  
*)
```

**5.4** Considere el tipo de dato `treeSk`, definido en Ocaml a continuación, que sirve para representar estructuras con forma de árbol ("esqueletos de árboles"), y las funciones `nodes` y `weight`, también definidas a continuación.

```
type treeSk = Tree of treeSk list;;
```

```
let rec nodes = function  
  | Tree [] -> 1  
  | Tree (h::t) -> nodes h + nodes (Tree t);;
```

```
let rec weight = function  
  | Tree [] -> 1  
  | Tree (h::t) -> 1+weight h + weight (Tree t);;
```

**5.4.1 (1.5 puntos)** Indique el efecto de la compilación y ejecución de las siguientes frases, como lo indicaría el compilador interactivo de Ocaml. Distinga claramente expresiones de definiciones.

```
nodes, weight;;
```

```
let t1 = Tree[];;
```

```
let t2 = Tree [t1] and t3 = Tree[t1;t1];;
```

```
let t4 = Tree [t2] and t5= Tree[t1;t3;t1;t1];;
```

```
let l = [t1;t2;t3;t4;t5] in List.map nodes l;;
```

```
let l = [t1;t2;t3;t4;t5] in List.map weight l;;
```