echo.ml

```
module S = Session.Bare

let echo_client ep x =
    let ep = S.send x ep in
    let res, ep = S.receive ep in
    S.close ep;
    res

let echo_service ep =
    let x, ep = S.receive ep in
    let ep = S.send x ep in
    S.close ep

let _ =
    let a, b = S.create () in
    let _ = Thread.create echo_service a in
    print_endline (echo_client b "Hello, world!")
```

echo_rec.ml

```
(*
val rec_echo_service : rec X.&[ End: end | Msg: ?α.!α.X ] → unit
val rec_echo_client : rec X.⊕[ End: end | Msg: !α.?β.X ] → α list → β list
*)

module S = Session.Bare

let rec rec_echo_service ep =
    match S.branch ep with
    | `Msg ep -> let x, ep = S.receive ep in
                 let ep = S.send x ep in
                 rec_echo_service ep
    | `End ep -> S.close ep

let rec rec_echo_client ep = function
    | []      -> let ep = S.select (fun x -> `End x) ep in
                 S.close ep;
                 []
    | x :: xs -> let ep = S.select (fun x -> `Msg x) ep in
                 let ep = S.send x ep in
                 let y, ep = S.receive ep in
                 y :: rec_echo_client ep xs

let _ =
    let a, b = S.create () in
    let _ = Thread.create rec_echo_service a in
    let res = rec_echo_client b ["uno"; "dos"; "tres"] in
    List.iter (Printf.printf "%s\n") res
```

ej1.ml

```
(*
val raiz : poly → float option
val linpoly_client : !poly.?float option → poly → unit
val linpoly_server : ?poly.!float option → unit
*)

module S = Session.Bare

type poly = {
    a: float; (* Coeficiente de x *)
    b: float; (* Termino constante *)
}

let raiz (p: poly): float option =
    if p.a = 0.0 then
        None
    else
        Some (-. p.b /. p.a)

let linpoly_client ep (p: poly) =
    let ep = S.send p ep in
    let r, ep = S.receive ep in

    match r with
    | Some v -> Printf.printf "%f\n" v
    | None -> Printf.printf "no hay raíz\n";

    S.close ep

let linpoly_server ep =
    let p, ep = S.receive ep in
    let ep = S.send (raiz p) ep in
    S.close ep

let _ =
  let a, b = S.create () in
  let _ = Thread.create linpoly_server a in
  linpoly_client b {a = 1.0; b = 2.0}
```

ej2.ml

```
(*
val raiz : poly → float option
val linpoly_client : !poly.&[ Raiz: ?float | SinRaiz: end ] → poly → unit
val linpoly_server : ?poly.⊕[ Raiz: !float | SinRaiz: end ] → unit
*)

module S = Session.Bare
```

```
type poly = {
    a: float; (* Coeficiente de x *)
    b: float; (* Termino constante *)
}

let raiz (p: poly): float option =
    if p.a = 0.0 then
        None
    else
        Some (-. p.b /. p.a)

let linpoly_client ep (p: poly) =
    let ep = S.send p ep in

    match S.branch ep with
    | `Raiz ep    -> let v, ep = S.receive ep in
                     Printf.printf "%f\n" v;
                     S.close ep
    | `SinRaiz ep -> Printf.printf "no hay raíz\n";
                     S.close ep

let linpoly_server ep =
    let p, ep = S.receive ep in
    let r = raiz p in

    match r with
    | Some v -> let ep = S.select (fun x -> `Raiz x) ep in
                let ep = S.send v ep in
                S.close ep
    | None   -> let ep = S.select (fun x -> `SinRaiz x) ep in
                S.close ep

let _ =
    let a, b = S.create () in
    let _ = Thread.create linpoly_server a in
    linpoly_client b {a = 1.0; b = 2.0}
```

ej3.ml

```
(*
val raiz : poly → float option
val linpoly_client : !float.!float.&[ Raiz: ?float | SinRaiz: end ] → poly → unit
val linpoly_server : ?float.?float.⊕[ Raiz: !float | SinRaiz: end ] → unit
*)

module S = Session.Bare

type poly = {
    a: float; (* Coeficiente de x *)
    b: float; (* Termino constante *)
}
```

```
let raiz (p: poly): float option =
    if p.a = 0.0 then
        None
    else
        Some (-. p.b /. p.a)

let linpoly_client ep (p: poly) =
    let ep = S.send p.a ep in
    let ep = S.send p.b ep in

    match S.branch ep with
    | `Raiz ep    -> let v, ep = S.receive ep in
                     Printf.printf "%f\n" v;
                     S.close ep
    | `SinRaiz ep -> Printf.printf "no hay raíz\n";
                     S.close ep

let linpoly_server ep =
    let a, ep = S.receive ep in
    let b, ep = S.receive ep in
    let r = raiz {a = a; b = b} in

    match r with
    | Some v -> let ep = S.select (fun x -> `Raiz x) ep in
                let ep = S.send v ep in
                S.close ep
    | None   -> let ep = S.select (fun x -> `SinRaiz x) ep in
                S.close ep

let _ =
    let a, b = S.create () in
    let _ = Thread.create linpoly_server a in
    linpoly_client b {a = 1.0; b = 2.0}
```

ej5.ml

```
(*
val raiz : poly → float option
val linpoly_client : !(?float.?float.⊕[ Raiz: !float | SinRaiz: end ]) → poly → unit
val linpoly_server : ?(?float.?float.⊕[ Raiz: !float | SinRaiz: end ]) → unit
*)

module S = Session.Bare

type poly = {
    a: float; (* Coeficiente de x *)
    b: float; (* Termino constante *)
}

let raiz (p: poly): float option =
    if p.a = 0.0 then
        None
    else
```

```
        Some (-. p.b /. p.a)

let linpoly_client ep (p: poly) =
    let a, b = S.create () in
    let ep = S.send b ep in
    let _ = S.close ep in

    let a = S.send p.a a in
    let a = S.send p.b a in

    match S.branch a with
    | `Raiz a    -> let v, a = S.receive a in
                    Printf.printf "%f\n" v;
                    S.close a
    | `SinRaiz a -> Printf.printf "no hay raíz\n";
                    S.close a

let linpoly_server ep =
    let c, ep = S.receive ep in
    let _ = S.close ep in

    let a, c = S.receive c in
    let b, c = S.receive c in
    let r = raiz {a = a; b = b} in

    match r with
    | Some v -> let c = S.select (fun x -> `Raiz x) c in
                let c = S.send v c in
                S.close c
    | None   -> let c = S.select (fun x -> `SinRaiz x) c in
                S.close c

let _ =
    let a, b = S.create () in
    let _ = Thread.create linpoly_server a in
    linpoly_client b {a = 1.0; b = 2.0}
```

ej6.ml

```
(*
val max_service : rec X.&[ End: !int | Num: ?int.X ] → int → unit
val max_client : rec X.⊕[ End: ?int | Num: !int.X ] → int list → int
*)

module S = Session.Bare

let rec max_service ep (m: int) =
    match S.branch ep with
    | `Num ep -> let x, ep = S.receive ep in
                 max_service ep (if x > m then x else m)
    | `End ep -> let ep = S.send m ep in
                 S.close ep
```

```
let rec max_client ep (nums: int list): int =
    match nums with
    | []      -> let ep = S.select (fun x -> `End x) ep in
                 let max, ep = S.receive ep in
                 S.close ep;
                 max
    | x :: xs -> let ep = S.select (fun x -> `Num x) ep in
                 let ep = S.send x ep in
                 max_client ep xs

let _ =
    let a, b = S.create () in
    let _ = Thread.create (max_service a) 0 in
    let res = max_client b [1;5;2;8;10;3;0;4] in
    Printf.printf "%d\n" res
```