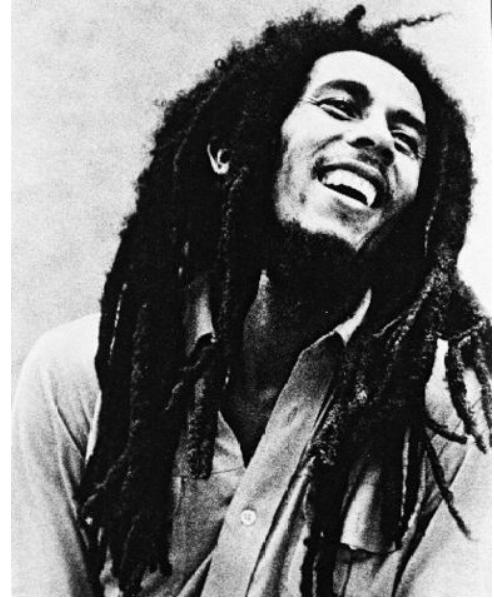

Lab3 - OCaml

— La Guerra Funcional —

Beta Ziliani

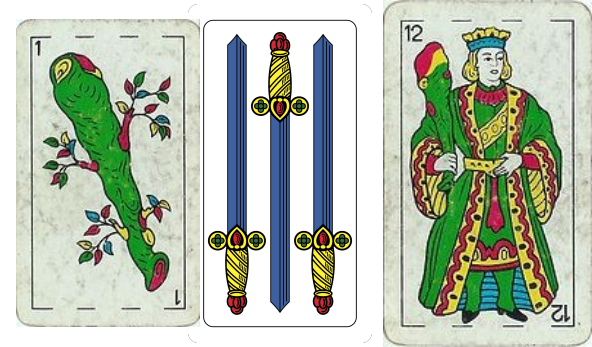
Alicia y Bob juegan "La Guerra Funcional"



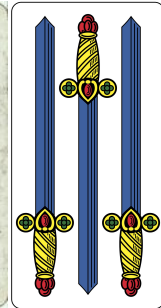
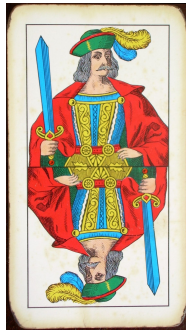
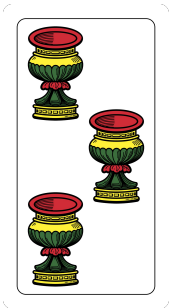
Alicia y Bob juegan "La Guerra Funcional"



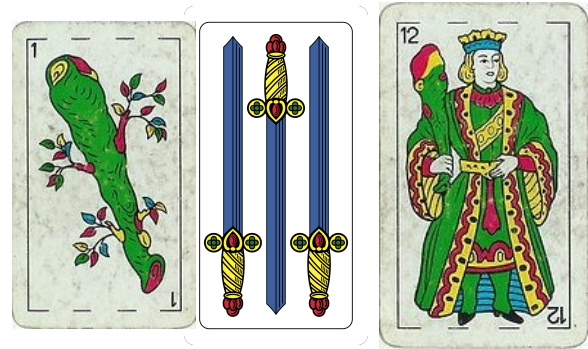
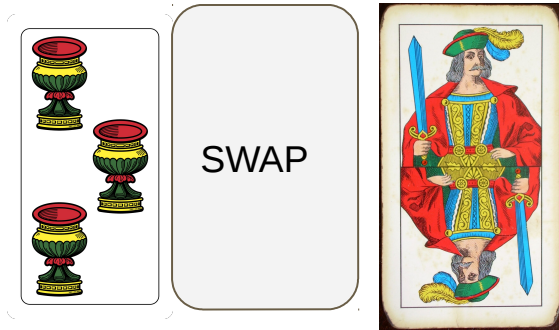
Alicia y Bob juegan "La Guerra Funcional"



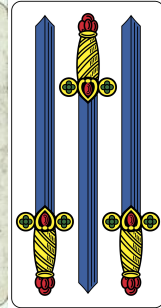
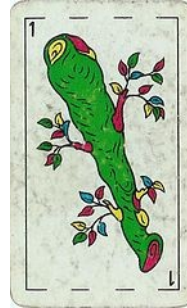
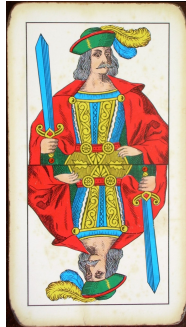
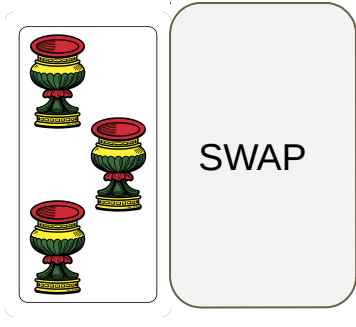
Alicia y Bob juegan "La Guerra Funcional"



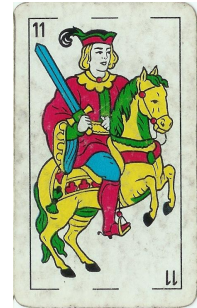
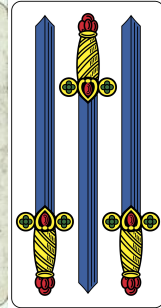
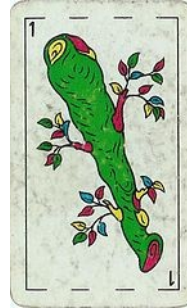
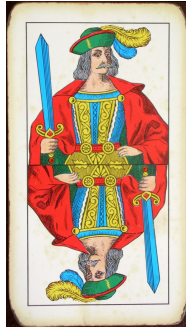
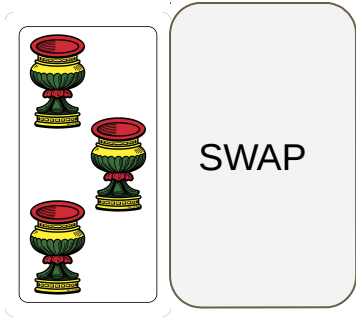
Alicia y Bob juegan "La Guerra Funcional"



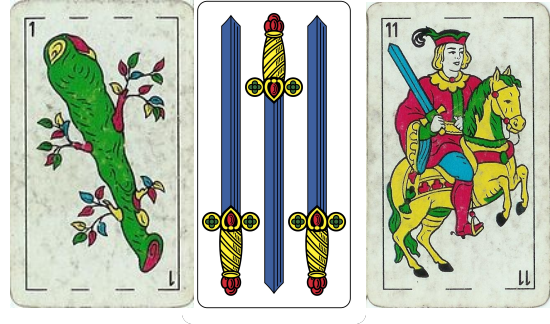
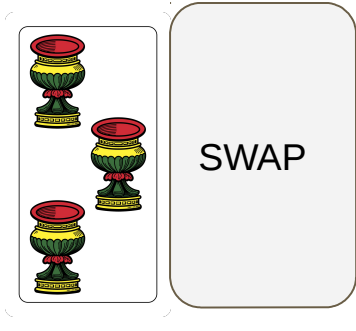
Alicia y Bob juegan "La Guerra Funcional"



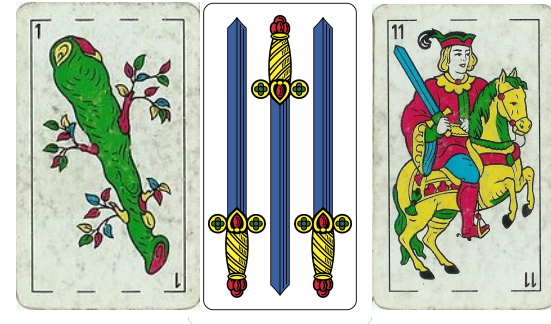
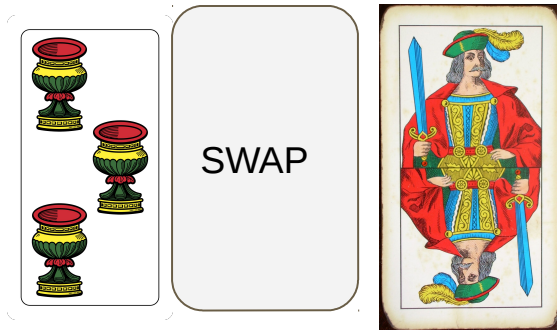
Alicia y Bob juegan "La Guerra Funcional"



Alicia y Bob juegan "La Guerra Funcional"



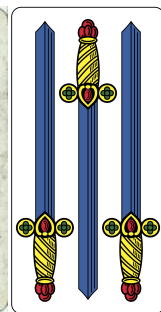
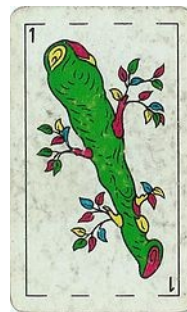
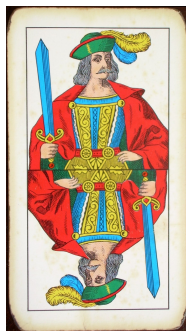
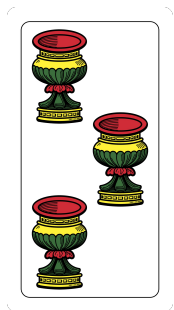
Alicia y Bob juegan "La Guerra Funcional"



Alicia y Bob juegan "La Guerra Funcional"



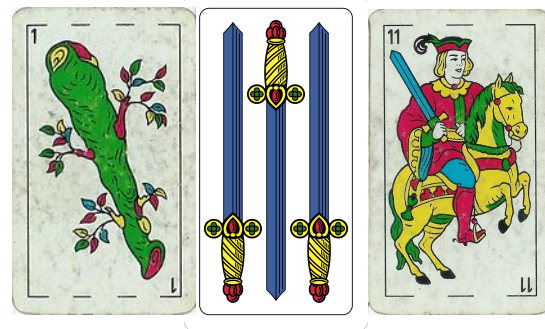
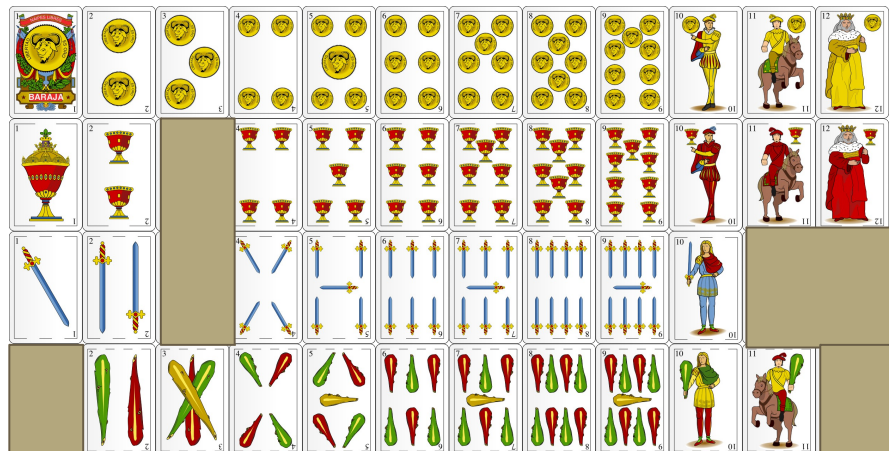
SWAP



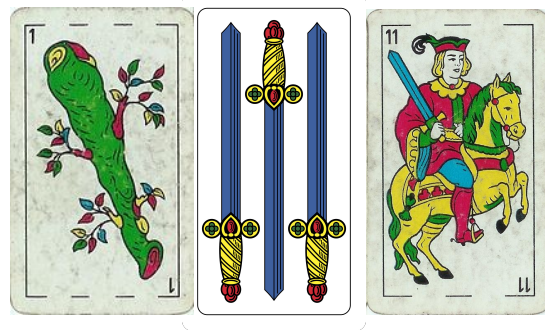
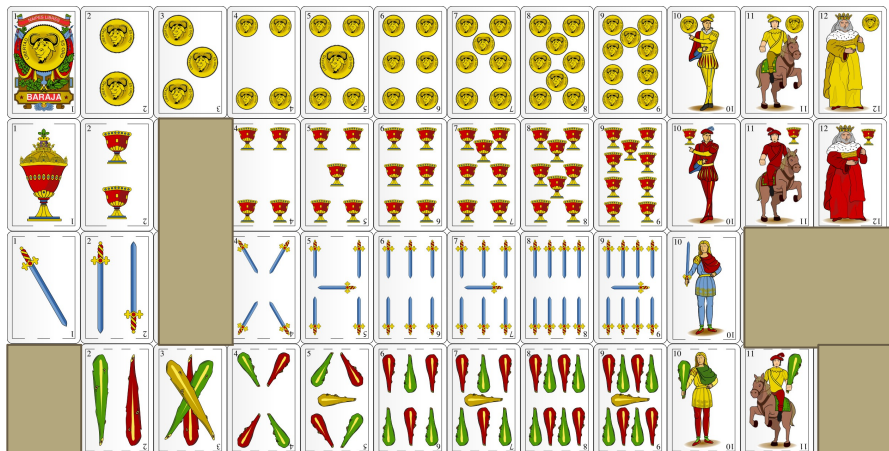
Alicia y Bob juegan "La Guerra Funcional"



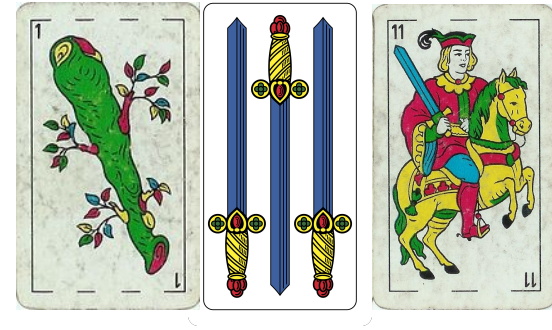
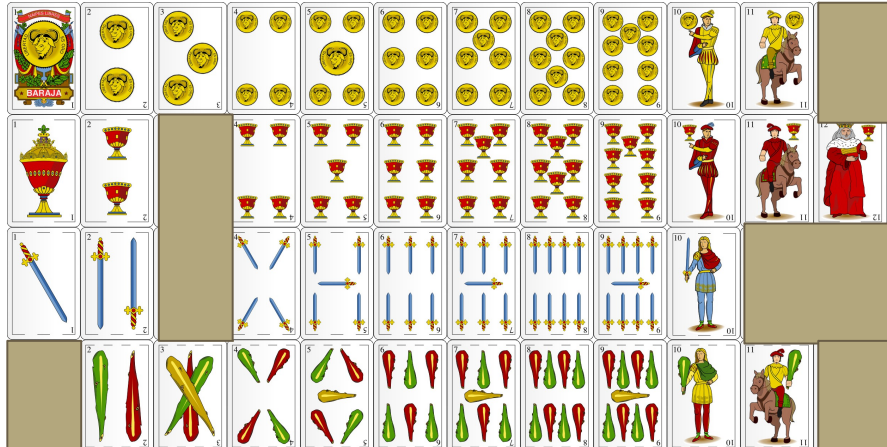
SWAP



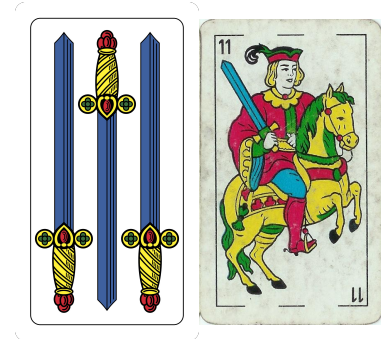
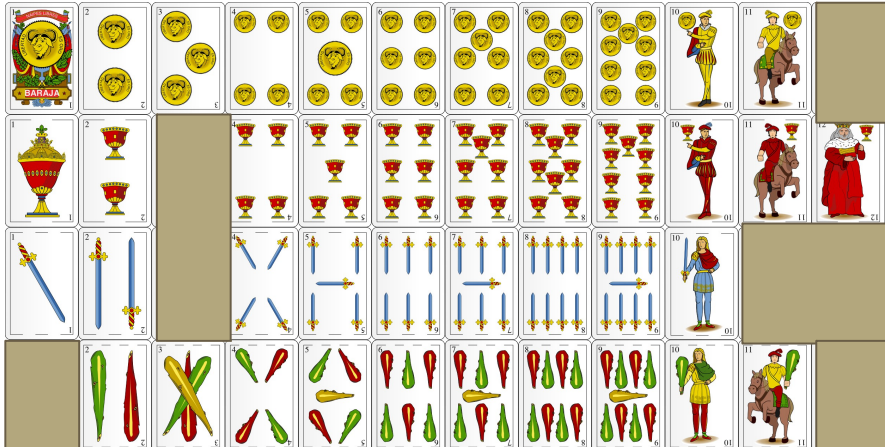
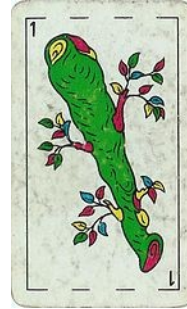
Alicia y Bob juegan "La Guerra Funcional"



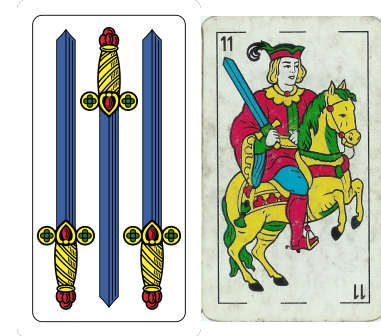
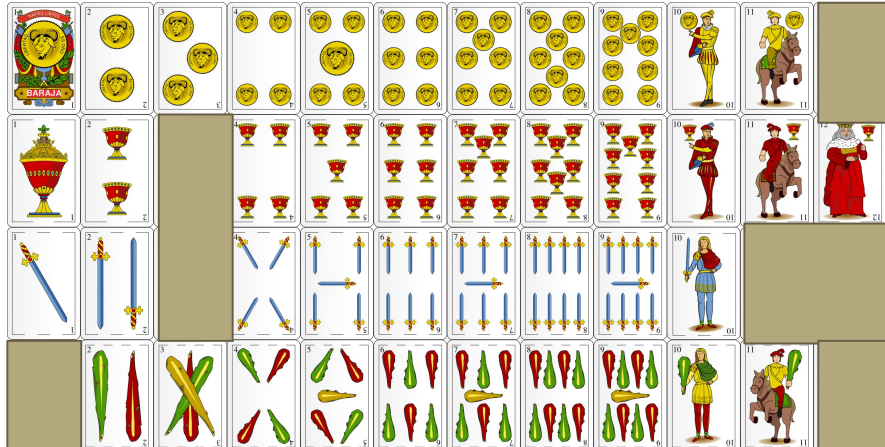
Alicia y Bob juegan "La Guerra Funcional"



Alicia y Bob juegan "La Guerra Funcional"



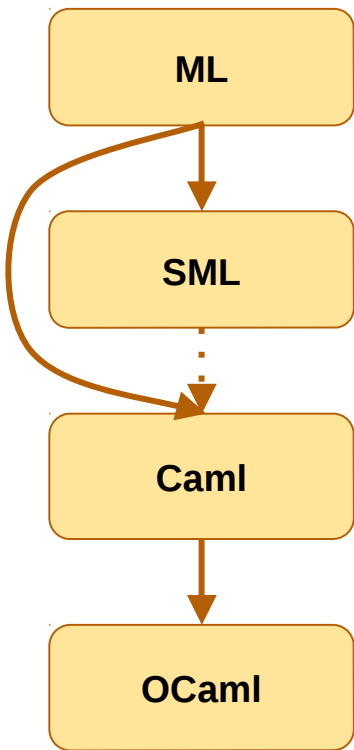
Alicia y Bob juegan "La Guerra Funcional"



Por qué OCaml?

- Por que ya vieron Haskell.
- Por que es mejor que Haskell.
- La verdad, por que es el mejor lenguaje.
- Posta.

Breve y simplificada reseña histórica



70s: Creado por **Robin Milner** para construir **LCF**.

80s: R. Milner; M. Tofte, **R. Harper** and D. MacQueen. **Módulos!** **SML/NJ** es el compilador más usado.

Early 90s: Ascander Suarez. Creado para construir a **Coq**.
Busca ser más cómodo de usar que SML. Utiliza **CAM**.

Late 90s: Xavier Leroy y Didier Rémy. **Objetos**.

Un ejemplito

```
(* archivo de interfaz hola.mli *)
```

```
type opaco
```

```
val a_imprimir : opaco
```

```
val imprimir : opaco -> unit
```

```
(* main.ml *)
```

```
open Hola
```

```
let _ = imprimir a_imprimir
```

```
(* archivo hola.ml *)
```

```
type opaco = UnEntero of int  
            | UnString of string
```

```
let a_imprimir = UnString "Hola mundo!"
```

```
let imprimir o =
```

```
  let open Printf in
```

```
  match o with
```

```
  | UnEntero n -> printf "%d\n" n
```

```
  | UnString s -> printf "%s\n" s
```

Un ejemplito

Ocultamiento

(* archivo de interfaz hola.mli *)
type opaco

val a_imprimir : opaco

val imprimir : opaco -> unit

(* main.ml *)
open Hola

let _ = imprimir a_imprimir

(* archivo hola.ml *)
type opaco = UnEntero of int
 | UnString of string

let a_imprimir = UnString "Hola mundo!"

let imprimir o =
 let open Printf in
 match o with
 | UnEntero n -> printf "%d\n" n
 | UnString s -> printf "%s\n" s

Un ejemplito

Importación

```
(* archivo de interfaz hola.mli *)
```

```
type opaco
```

```
val a_imprimir : opaco
```

```
val imprimir : opaco -> unit
```

```
(* main.ml *)
```

```
open Hola
```

```
let _ = imprimir a_imprimir
```

```
(* archivo hola.ml *)
```

```
type opaco = UnEntero of int  
            | UnString of string
```

```
let a_imprimir = UnString "Hola mundo!"
```

```
let imprimir o =
```

```
  let open Printf in
```

```
  match o with
```

```
  | UnEntero n -> printf "%d\n" n
```

```
  | UnString s -> printf "%s\n" s
```

Un ejemplito

```
(* archivo de interfaz hola.mli *)
```

```
type opaco
```

```
val a_imprimir : opaco
```

```
val imprimir : opaco -> unit
```

```
(* main.ml *)
```

```
open Hola
```

```
let _ = imprimir a_imprimir
```

Pattern match

```
(* archivo hola.ml *)
```

```
type opaco = UnEntero of int  
           | UnString of string
```

```
let a_imprimir = UnString "Hola mundo!"
```

```
let imprimir o =
```

```
  let open Printf in
```

```
  match o with
```

```
  | UnEntero n -> printf "%d\n" n
```

```
  | UnString s -> printf "%s\n" s
```

ocamlbuild



- Compilar es una papa!

- Si los archivos están en **./src** :

```
> ocamlbuild -I src main.native
```

- Mágicamente encuentra dependencias.

- Pone todos los archivos **.o** en **./_build**.

- Igual van a necesitar un **Makefile** y mover a mano el ejecutable a **./bin** !

Debuggear

- Compilar para debugging:

```
> ocamlbuild -I src main.d.byte
```

- Debuggear como con **gdb**:

```
> ocamldebug main.d.byte
```

- Luego

- run
- break Hola.imprimir
- reverse
- step
- ...

GUI (editor)?

- Emacs
- Vim
- Atom
- Eclipse
- ...

La pureza es para los ñoños

- En **Haskell** imprimir algo en pantalla o guardar algo en memoria requiere de "**mónadas**" y "**type classes**".
- En OCaml se puede hacer simple:

```
let double n =  
  let r = ref n in  
  r := r + n;  
  Printf.printf "%d\n" !r;  
  !r
```



La pureza es para los ñoños

● También tiene **excepciones**:

`exception NotEmpty`

```
let tail s =  
  match s with  
  | (x :: s') -> s'  
  | _ -> raise NotEmpty
```



Pero qué somos nosotros?

Para el Lab eviten usar refs!

Y solo excepciones en casos...

**excepcionales! Prefieran el tipo
option.**

Es un labo de *funcional*.



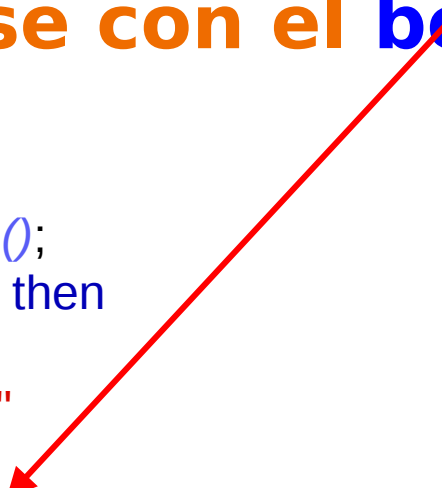
Amíguense con el begin - end

```
let _ =  
  Random.self_init ();  
  if Random.bool () then  
    printf "hola";  
    printf " mundo\n"  
  else  
    printf "chau";  
    printf " mundo\n"
```

Amíguense con el **be**nd

Syntax Error!

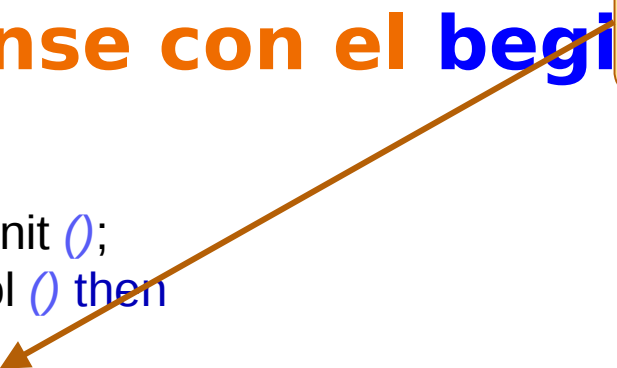
```
let _ =  
  Random.self_init ();  
if Random.bool () then  
  printf "hola";  
  printf " mundo\n"  
else  
  printf "chau";  
  printf " mundo\n"
```



Amíguense con el **begin**

Bloque begin-end

```
let _ =  
  Random.self_init ();  
  if Random.bool () then  
    begin  
      printf "hola";  
      printf " mundo"  
    end  
  else  
    begin  
      printf "chau";  
      printf " mundo\n"  
    end
```



Amígense con el begin - end

```
let _ =  
  Random.self_init ();  
  if Random.bool () then  
    printf "hola"  
  else  
    printf "chau";  
  printf " mundo\n"
```


Viviendo en las sombras

```
let x = 0
```

```
let a_function x1 =  
  let x2 = x1 + 1 in  
  let x3 = x2 + 1 in  
  x3
```

```
let two = a_function x
```

- Este código es propenso a errores.



Viviendo en las sombras

```
let x = 0
```

```
let a_function x =  
  let x = x + 1 in  
  let x = x + 1 in  
  x
```

```
let two = a_function x
```

- En lenguajes como ML es común resignificar una variable.



Records: Una de las virtudes de OCaml

```
type account = { first : string; last : string; balance : float }
```

```
let beta1 = {last="Ziliani"; first="Beta"; balance=1000000.0}
```

```
let beta2 = {beta1 with balance=5.0}
```

```
let name {first=f; last=l; _} = f ^ " " ^ l
```

```
let _ = printf "%s: %f %f\n" (name beta1) beta1.balance beta2.balance
```

Records: Una de las virtudes de

Copia

```
type account = { first : string; last : string; balance : float }
```

```
let beta1 = {last="Ziliani"; first="Beta"; balance=1000000.0}
```

```
let beta2 = {beta1 with balance=5.0}
```

```
let name {first=f; last=l; _} = f ^ " " ^ l
```

```
let _ = printf "%s: %f %f\n" (name beta1) beta1.balance beta2.balance
```

Records: Una de las virtudes de OCaml

Pattern match

```
type account = { first : string; last : string; balance : float }
```

```
let beta1 = {last="Ziliani"; first="Beta"; balance=1000000.0}
```

```
let beta2 = {beta1 with balance=5.0}
```

```
let name {first=f; last=l; _} = f ^ " " ^ l
```

```
let _ = printf "%s: %f %f\n" (name beta1) beta1.balance beta2.balance
```

OCaml es mucho más...



OCaml

- No vamos a utilizar **objetos**, ni **functores**, ni **módulos recursivos**, ni ...

Información:

- <http://ocaml.org/learn/tutorials/> (Tutoriales varios)
- <http://caml.inria.fr/pub/docs/manual-ocaml/> (Manual)
- <http://www.cs.cornell.edu/courses/cs3110/2011sp/lecturenotes.asp> (Curso)
- <https://realworldocaml.org/> (Libro. OJO! Utiliza la librería Core y no STD)
- <http://structio.sourceforge.net/guias/progocaml/progocaml.html> (En español, no se qué tan actualizado)
- <https://try.ocamlpro.com/> (OCaml en el navegador)