8 - Imperative OCaml

florian.felten@uni.lu

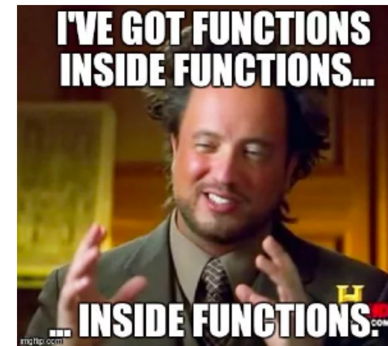
UNIVERSITÉ DU LUXEMBOURG

Slides inspired from Pierre Talbot.

I feel you

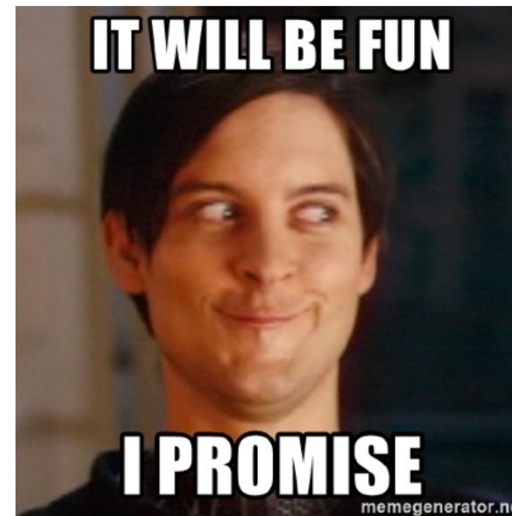
“I just don’t understand why someone would use a language with variables which are actually constants”
- Probably you.

“FP is pretty cool, I am going to use it for sure”
- Probably you at the exam.



The promise

- You will see that we can in fact mutate the memory in OCaml.
- There are even mutable data structures!
- Exceptions are also doable in OCaml.
- We will build simple programs using that.



By the end of this course, my promise is that you…

Imperative OCaml (K_?)

$< D > ::=$ Type declaration

$\mid \ldots$	
$\mid$ exception e of T	(exception)

Imperative statements are expressions which return unit i.e. they don't return

$< n, m, p, q, \ldots > ::=$ Expressions

$\mid \ldots$	
$\mid$ raise e	(raise an exception)
$\mid$ try p with e -> q	(catch an exception)
$\mid$ for $i = n$ [to | down to] m do p done	(for loop)
$\mid$ while b do p done	(while loop)
$\mid$ n <- m	(update operator)

Exceptions

```
exception  EmptyList (* Exception type declaration *)

let head = function

   | [] -> raise EmptyList (* Raise an exception *)

   | a::_ -> a

in

let _ =

   try head []  (* statement which might raise an exception *)

   with EmptyList -> Printf.printf “The list is empty.\n” (* Catch an exception *)
```

Sequencing statements

We can sequence imperative statements (returning unit) with the “;” operator:

```
let _ = Printf.printf “hello” in
Printf.printf “world”
```

Becomes:

```
Printf.printf “hello”;
Printf.printf “world”
```

This says “ignore the value returned by this line”

q: Why do I say that imperative statements return unit ?

What does this do ?

```
let print_hello name =
   Printf.printf "hello";
   Printf.printf "%s\n" name;
let _ = print_hello "you"
```

Careful with sequences

The compiler might not understand what you mean!

```
let print_hello name =
   Printf.printf “hello”;
   Printf.printf “%s\n” name;
   let _ = print_hello “you”
```

This is what the compiler understands, which generates not so nice syntactic error messages.

Solutions?

Use **parenthesis** to group sequenced statements:

```
let print_hello name =
    (Printf.printf "hello";
    Printf.printf "%s\n" name;)
in
print_hello "you";
```

Use **begin** statement:

```
let print_hello name =
    begin
        Printf.printf "hello";
        Printf.printf "%s\n" name;
    end
in
print_hello "you";
```


…Don't use imperative stuff?

(mutable) Arrays

Arrays

Resources: https://v2.ocaml.org/api/Array.html

Some operations:

```
get: 'a array -> int -> 'a
```

`Array.get a n` returns the element number `n` of array `a`.

```
set: 'a array -> int -> 'a -> unit
```

`Array.set a n x` modifies array `a` in place, replacing element number `n` with `x`

```
length: 'array -> int
```

```
make: int -> 'a -> 'a array
```

`Array.make n x` returns a fresh array of length `n`, initialized with `x`.

~ your beloved Python lists.

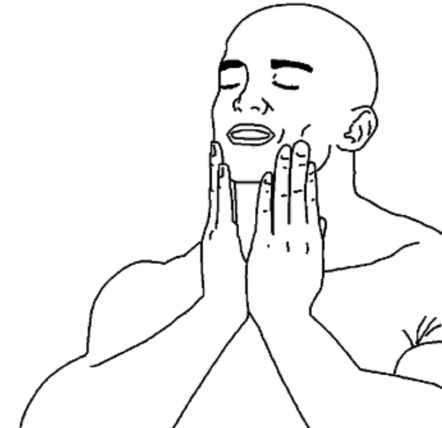
Some (syntactic) sugar !

```
let t = [|1; 2; 4|] in

…

Array.get t i        =        t.(i)

Array.set t i v      =        t.(i) <- v
```



Loops

The body of a loop should have type unit…

and usually cause side effects e.g. mutation of something

```
for i=10 downto 0 do
 Printf.printf "%d " i;
done
```

```
let t = [|1; 2|] in
for i=0 to (Array.length t) - 1 do
 t.(i) <- t.(i) + 1;
done
```

```
(* Functional style on array *)
let t1 = Array.map (fun n -> n + 1) t in  …
```

Mutable records

Records

```
(* Type declaration *)

type coloredPoint = { mutable xy: coord; c: string };;


(* Update *)

let p = { xy={x=3; y=2}; c="Red" } in

begin

   p.xy <- {x=5; y=4};

   Printf.printf "(%d, %d, %s)" p.xy.x p.xy.y p.c

end
```

Mutable variables

Using mutable records, we can have mutable variables, as in all imperative languages

```
type cell = {mutable  content: int};;
```

There you go:

```
(* OCaml *)

let i = {content=0} in

i.content <- (i.content + 1)
```

```
# Python

i=0

i+=1
```

A bit of boilerplate when compared to Python… we can do better!

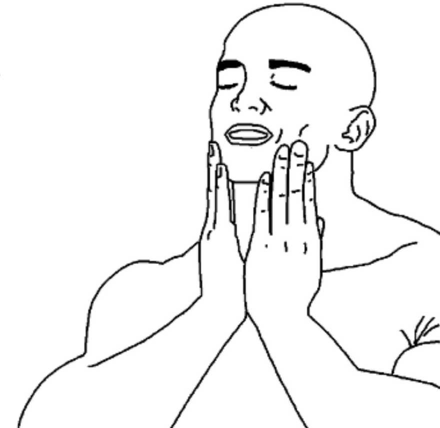
Ref type

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

ref: 'a -> 'a ref

(!): 'a ref -> 'a (* Access *)

(:=): 'a ref -> 'a -> unit (*Mutation *)
```



```
let i = ref 0 in

i := !i + 1
```

Resource: https://cs3110.github.io/textbook/chapters/mut/refs.html

Take aways - did I hold my promise?

- ✔ We saw how to handle mutations in OCaml (variables, loops, arrays)
- ✔ We saw that exceptions can be used too!
- ✔ Now that Pierre is gone, we can fall back to proper imperative style, can't we? 😈

Exercises