Prof : letrouzey@irif.fr

La programmation fonctionnelle utilise des mécanismes totalement différents.

Exemple : tri3 : liste → liste (= fonction)

- tri3(∅) = ∅
- tri3(x puis l) = tri3(l filtrée par < x)) puis x puis tri3(l filtrée par (≥ x))

A.N : tri3(1, 3, 2, 5)

↳ tri3(∅) 1 tri3(3, 2, 5)

↳ ∅ 1 tri3(∅) 3 tri3(2, 5)

↳ ⋮

↳ ⋮

↳ 1 2 3 5

On fait des remplacement d'expressions

- tri3 ne modifie pas
  ↳ elle CONSTRUIT
- tri3 est définie par induction

Remarque : Garbage collector en Ocaml

Définition : Une fonction est pure ssi elle ne modifie pas son environnement

$$\forall x, y \text{ tq } x = y,\ F(x) = F(y)$$

POO c'est "Modulaire par classes"
Fonctionnel c'est "instancier différemment l'existant"

En Ocaml, l'opération `x :: xs` fait `[1, 2, ..., n]`

`x` `[xs]`

Exemples de code :

Sites pour coder : try.ocamlpro.com
sketch.sh

Dans un

```
if ...
then
  type1
else
  type2
```

type1 et type2 doivent être égaux

- Le type string existe.

"abc" ^ "def" donne "abcdef" et "abc".[0] donne 'a'

- Char avec ('a') simples guillemets

Si on a

```
let x = 5 ;;
→ 5
let x = 10 in x + x ;;
→ 20
x ;;
→ 5
```

Rend le x précédent

LOCAL

Donc

```
let x = 5 ;;
let z = 2 in x * z ;;
z
```

Ne fonctionne pas.

Les commentaires s'encadrent avec (* ... *)

Ex: (* Description *)

Les Fonctions :

Introduites par fun

```
fun x -> 2*x ;;
```

Comme une fonction anonyme / lambda

→ Fonction double.

Mais on ne peut pas l'appeler...
Sauf si on l'applique directement,

```
(fun x -> 2*x)(10)
→ 20
```

Pour réutiliser : argument(s) ; expression retournée

```
let double x = 2*x ;;
```

Ou

```
let double = fun x -> 2*x
```

nom ; déclaration

et l'appel est :

```
double (10)        ou        double 10
→ 20                         → 20
```

Si l'arg est une constante positive/variable, on peut choisir la deuxième façon
Sinon (ex `double 10+10 ;;`)
→ 30 et non 40
on écrirait `double (10+10)`

En pratique on parenthèse partout

```
if true
  then
    fun x -> 2*x
  else
    fun x -> x / 2
```

est un code valable

Attention la "vraie" comparaison est un simple égal...

la comparaison des pointeurs est ==

La récursivité se note avec un rec

```
let rec fact n =
  if n = 0
    then
      1
  else
    fact (n-1) * n
```