

# Programmation fonctionnelle avancée

Notes de cours

Mise à niveau 1

9 septembre 2015

Sylvain Conchon

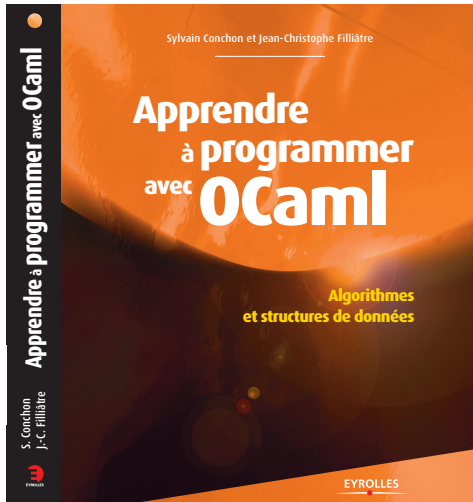
`sylvain.conchon@lri.fr`

Le cours est à la fois :

- ▶ Une introduction à la programmation fonctionnelle
- ▶ Une initiation au langage OCaml

Un livre qui peut vous aider (Amazon, FNAC, etc.)

<http://programmer-avec-ocaml.lri.fr/>



# Environnements de travail

<http://caml.inria.fr>

- ▶ langage développé à l'**INRIA** (Institut National de Recherche en Informatique et en Automatique)
- ▶ disponible sur de nombreuses architectures (Linux, Windows, Mac OS X etc.)

Nous allons utiliser les outils suivants en TP :

- ▶ un **terminal** `unix`
- ▶ un **éditeur de texte** : `emacs`
- ▶ un **interpréteur de commandes OCaml** : `ocaml`
- ▶ un **compilateur** pour OCaml : `ocamlc`

# Démonstration

Le premier programme :  
afficher `Hello world!` à l'écran

# L'interpréteur : lancement

Très pratique pour écrire de petits programmes ou connaître la valeur (ou le type) d'une expression

Pour cela, dans la terminal je tape simplement `ocaml`

```
> ocaml
      OCaml version 4.01.0
#
```

Le symbole `#` est l'**invite de commande** de l'interpréteur : il vous invite à écrire une expression OCaml



# L'interpréteur : évaluation d'une expression

L'évaluation d'une expression se fait en **trois temps**

```
> ocaml
  OCaml version 4.01.0
#
```

# L'interpréteur : évaluation d'une expression

L'évaluation d'une expression se fait en **trois temps**

```
> ocaml  
OCaml version 4.01.0  
# 1 + 2;;
```

1. Je **saisie** l'expression et j'indique que j'ai terminé en tapant **;;**

# L'interpréteur : évaluation d'une expression

L'évaluation d'une expression se fait en **trois temps**

```
> ocaml  
OCaml version 4.01.0  
# 1 + 2;;
```

1. Je **saisie** l'expression et j'indique que j'ai terminé en tapant **;;**
2. L'interpréteur **évalue** mon expression

# L'interpréteur : évaluation d'une expression

L'évaluation d'une expression se fait en **trois temps**

```
> ocaml
OCaml version 4.01.0
# 1 + 2;;
- : int = 3
```

1. Je **saisie** l'expression et j'indique que j'ai terminé en tapant **;;**
2. L'interpréteur **évalue** mon expression
3. Puis il **affiche** son type et sa valeur

# L'interpréteur : évaluation d'une expression

L'évaluation d'une expression se fait en **trois temps**

```
> ocaml
OCaml version 4.01.0
# 1 + 2;;
- : int = 3
#
```

1. Je **saisie** l'expression et j'indique que j'ai terminé en tapant **;;**
2. L'interpréteur **évalue** mon expression
3. Puis il **affiche** son type et sa valeur

# Le compilateur : édition, compilation, exécution

L'écriture de (gros) programmes nécessite un éditeur et un compilateur

Le cycle d'utilisation du compilateur est également en **trois temps**

# Le compilateur : édition, compilation, exécution

L'écriture de (gros) programmes nécessite un éditeur et un compilateur

Le cycle d'utilisation du compilateur est également en **trois temps**

1. J'écris mon programme `hello.ml` à l'aide d'`emacs`

# Le compilateur : édition, compilation, exécution

L'écriture de (gros) programmes nécessite un éditeur et un compilateur

Le cycle d'utilisation du compilateur est également en **trois temps**

1. J'écris mon programme `hello.ml` à l'aide d'`emacs`

2. Dans le terminal, je compile

```
> ocamlc -o hello hello.ml  
>
```



# Le compilateur : édition, compilation, exécution

L'écriture de (gros) programmes nécessite un éditeur et un compilateur

Le cycle d'utilisation du compilateur est également en **trois temps**

1. J'écris mon programme `hello.ml` à l'aide d'`emacs`

2. Dans le terminal, je compile

```
> ocamlc -o hello hello.ml  
>
```

3. J'exécute mon programme

```
> ./hello  
Hello wolrd!
```

# Programme 1 : Années bissextiles

Notions introduites :

- ▶ forme générale d'un programme
- ▶ types de base (`int`, `bool`, `string`, `unit`)
- ▶ construction `let`
- ▶ appel de fonction
- ▶ fonction d'affichage `Printf.printf`

# La forme des programmes Ocaml

Un programme OCaml est simplement :

- ▶ une suite de déclarations (`let`) ou d'expressions à évaluer (de haut en bas)
- ▶ la fin d'une déclaration ou d'une expression est spécifiée par deux points virgules `;;`

Il n'y a donc pas de point d'entrée particulier (fonction principale par ex.) comme dans d'autres langages.

# Types et expressions élémentaires

Expressions de type `int` : les entiers

```
# 4 + 1 - 2 * 2;;  
- : int = 1  
# 5 / 2;;  
- : int = 2  
# 1_000_005 mod 2;;  
- : int = 1  
# max_int + 1;;  
- : int = -1073741824
```

- ▶ `int` représente les entiers compris entre  $-2^{30}$  et  $2^{30} - 1$  (sur une machine 32 bits)
- ▶ opérations sur ce type : `+`, `-`, `*`, `/` (division entière), `mod` (reste de la division) etc.

# Types et expressions élémentaires

Expressions de type `bool` : les valeurs booléennes

```
# false || true ;;
- : bool = true
# 3 <= 1 ;;
- : bool = false
# not (0=2) && 1>=3 ;;
- : bool = false
# if 2<0 then 2.0 else (4.6 *. 1.2) ;;
- : float = 5.52
```

- ▶ les constantes `true` (vrai) et `false` (faux)
- ▶ les opérations sur ce type `not` (non), `&&` (et) et `||` (ou)
- ▶ les opérateurs de comparaison (`=`, `<`, `>`, `<=`, `>=`) retournent des valeurs booléennes
- ▶ dans une conditionnelle de la forme

`if exp1 then exp2 else exp3`

l'expression `exp1` doit être de type `bool`.

Expressions de type `char` : les caractères

```
# 'a' ;;  
- : char = 'a'  
# int_of_char 'a' ;;  
- : int = 97  
# char_of_int 100 ;;  
- : char = 'd'
```

- ▶ les caractères sont encadrés par deux apostrophes '
- ▶ la fonction `int_of_char` renvoie le code ASCII d'un caractère (et inversement pour la fonction `char_of_int`).

# Types et expressions élémentaires

Expressions de type `string` : les chaînes de caractères

```
# "hello" ;;  
- : string = "hello"  
# "" ;;  
- : string = ""  
# "bon" ^ "jour" ;;  
- : string = "bonjour"  
# "hello".[1] ;;  
- : char = 'e' # string_of_int 123 ;;  
- : string = "123"
```

- ▶ ces valeurs sont encadrées par deux guillemets `"`
- ▶ l'opérateur `^` concatène des chaînes
- ▶ l'opération `.[i]` accède au *i*<sup>e</sup> caractère d'une chaîne (le premier caractère est à l'indice 0)
- ▶ des fonctions de conversions permettent de convertir des valeurs de types de base en chaînes (et inversement)

# Le type unit

```
# () ;;  
- : unit = ()  
# Print.printf "bonjour\n" ;;  
bonjour  
- : unit = ()
```

- ▶ unit représente les expressions qui font uniquement des effets de bord
- ▶ une seule valeur a ce type, elle est notée **()**
- ▶ c'est l'équivalent du type void en C



# Les variables globales

```
# let x = 3;;  
val x : int = 3
```

- ▶ le type est **inféré** automatiquement par le compilateur
- ▶ le contenu d'une variable **n'est pas modifiable**
- ▶ la portée est limitée aux déclarations suivantes

```
# let y = 5 + x;;  
val y : int = 8  
# let z = 10 + z;;  
Error: Unbound value z
```

- ▶ La liaison est **statique** : la redéfinition ne change pas la valeur des expressions précédentes

```
# let x = 10;;  
val x : int = 10  
# y;;  
- : int = 8
```

# Appel de fonction

L'appel d'une fonction `toto` avec un argument `v` se note simplement :

```
toto v
```

il n'y a donc pas de parenthèses

Si la fonction `toto` a plusieurs arguments, par exemple trois arguments, on note simplement :

```
toto v1 v2 v3
```

# La fonction `Printf.printf`

(Il s'agit d'une fonction très connue des programmeurs C)

`Printf.printf` prend comme premier argument une chaîne de formatage qui contient le message à écrire

Ce message peut contenir des codes de format (commençant par %) qui indiquent qu'un argument est attendu à cet endroit

Selon le code de format, le type de l'argument est :

- `%d` un entier
- `%c` un caractère
- `%f` un flottant
- `%s` une chaîne de caractères

La fonction `Print.printf` attend donc autant d'arguments que nécessaire pour construire le message à afficher

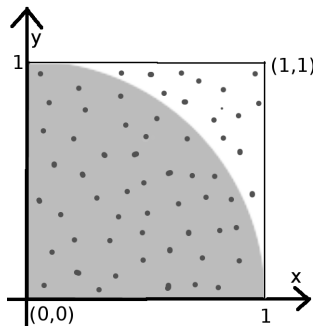
## Programme 2 : Méthode de Monte-Carlo

Notions introduites :

- ▶ nombres flottants (type `float`)
- ▶ variables locales (construction `let-in`)
- ▶ bibliothèque de nombres aléatoires `Random`
- ▶ accès aux arguments d'un programme (`Sys.argv`)
- ▶ déclaration de fonctions
- ▶ fonctions récursives
- ▶ fonctions à plusieurs arguments

# Calcul de $\pi$ par la méthode de Monte-Carlo

- ▶ soit un carré de côté 1 et le quart de cercle de rayon 1 inscrit dans ce carré (l'aire de ce cercle est  $\pi/4$ )
- ▶ si l'on choisit au hasard un point du carré, la probabilité qu'il soit dans le quart de cercle est donc également de  $\pi/4$
- ▶ en tirant au hasard un grand nombre  $n$  de points dans le carré, si  $p$  est le nombre de points à l'intérieur du cercle, alors  $4 \times p/n$  donne une bonne approximation de  $\pi$



# Le type float

Expressions de type `float` : les nombres à virgule flottante

```
# 4.3e4 +. 1.2 *. -2.3 ;;  
- : float = 42997.24  
# 5. /. 2. ;;  
- : float = 2.5  
# 1. /. 0. ;;  
- : float = infinity  
# 0. /. 0. ;;  
- : float = nan
```

- ▶ les types `int` et `float` sont disjoints
- ▶ opérations sur ce type : `+. -. *. /. sqrt cos` etc.
- ▶ on passe d'un entier à un flottant à l'aide de la fonction `float_of_int` et inversement avec `truncate`

# Les variables locales

```
# let x = 3 in x + 1;;  
- : int = 4
```

- la portée est limitée à l'expression qui suit le **in**

```
# x + 2;;  
Error: Unbound value x
```

- le nom de la variable locale masque toute déclaration antérieure de même nom

```
# let y = 2;;  
val y : int = 2  
# let y = 100.5 in (truncate y) + 1;;  
- : int = 101  
# y + 3;;  
- : int = 5
```

# La bibliothèque Random

Cette bibliothèque contient plusieurs fonctions pour générer des nombres aléatoires

`Random.float n` renvoie, de manière aléatoire, un nombre flottant entre 0 et `n` (inclus) (si `n` est négatif, le résultat est négatif ou 0)

Il existe d'autres fonctions comme `Random.int` (pour générer des entiers), `Random.bool`, etc.

Il faut appeler la fonction `Random.self_init ()` pour initialiser le générateur (sans quoi on obtient toujours la même séquence pour chaque exécution)



# Les arguments d'un programme

L'accès aux arguments d'un programme (passés sur la ligne de commande dans le terminal) se fait à l'aide de la notation

`Sys.argv.(i)`

où `i` est un entier tel que :

- 0 le nom du programme exécuté
- 1 le premier argument
- 2 le deuxième argument
- ...

`Sys.argv.(i)` est une **chaîne de caractères**

Ainsi, si je tape dans le terminal

```
> ./approx_pi 100
```

`Sys.argv.(0)` vaut `"/approx"`, et `Sys.argv.(1)` vaut `"100"`

# Fonctions

```
# let f x = x + 2;;  
val f : int -> int = <fun>  
# f 4;;  
- : int = 6
```

- les types, **des arguments et du résultat**, sont inférés
- la règle de portée du nom de la fonction est identique à celle des constantes (globales ou locales)

```
# let h x = x / 2 in h 6;;  
- : int = 3  
# h 4;;  
Error : Unbound value h  
# let g x = x || g (not x);;  
Error : Unbound value g
```

# Fonctions à plusieurs arguments

```
# let f x y z =  
    if x then y + 1 else z - 1;;  
val f : bool -> int -> int -> int = <fun>  
# f true 2 3;;  
- : int = 3
```

- les paramètres ne sont pas entre parenthèses, ni dans les déclarations, ni dans les applications de fonctions

# Fonctions récursives

```
# let rec fact x =  
    if x <= 0 then 1  
    else x * fact (x - 1);;  
val fact : int -> int = <fun>  
# fact 4;;  
- : int = 24
```

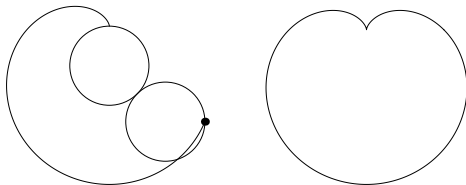
- l'ajout du mot-clé **rec** change la portée de l'identificateur : il est alors accessible dans la définition de la fonction

## Programme 3 : Dessin d'une cardioïde

Notions introduites :

- ▶ bibliothèque `Graphics`
- ▶ Séquence d'expressions

# Définition d'une cardioïde

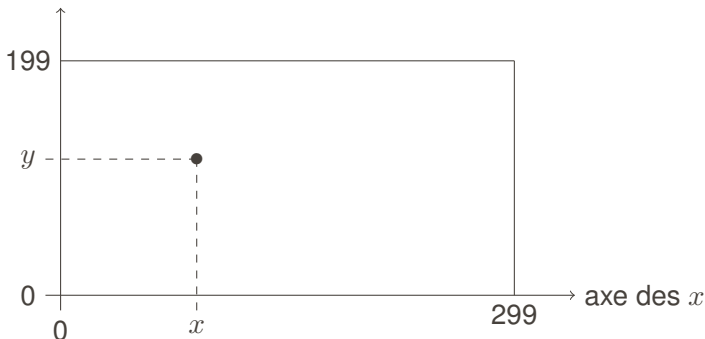


$$\begin{cases} x(\theta) = a(1 - \sin(\theta)) \cos(\theta) \\ y(\theta) = a(1 - \sin(\theta)) \sin(\theta) \end{cases}$$

# La bibliothèque Graphics

- ▶ ajouter le fichier `graphics.cmx` dans la commande de compilation (`ocamlc ... graphics.cmx ...`)
- ▶ la directive `open Graphics` permet d'accéder directement aux fonctions de la bibliothèque `Graphics`
- ▶ `open_graph " 300x200"` ouvre une fenêtre graphique de 300 pixels de large et 200 de haut

axe des  $y$



# Quelques fonctions de Graphics

- ▶ `plot x y` affiche un pixel en  $(x, y)$  dans la couleur courante
- ▶ `rgb r v b` renvoie une couleur calculée à partir de composantes rouge, vert et bleu (entiers entre 0 et 255) ; les couleurs prédéfinies : `white`, `black`, `blue`, `green` etc.
- ▶ `set_color c` fixe la couleur courante à la valeur `c`
- ▶ `let st = wait_next_event [Button_down]` attend un clic de souris ; les coordonnées sont `st.mouse_x` et `st.mouse_y`



# Séquences

Une séquence d'expressions permet d'évaluer des expressions **les unes après les autres**

```
# let x = Print.printf "bonjour\n"; 5;;  
bonjour  
val x : int = 5
```

- ▶ l'opérateur de séquence est le point virgule ;
- ▶ c'est un opérateur **binaire**

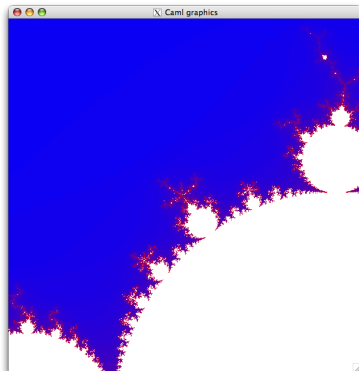
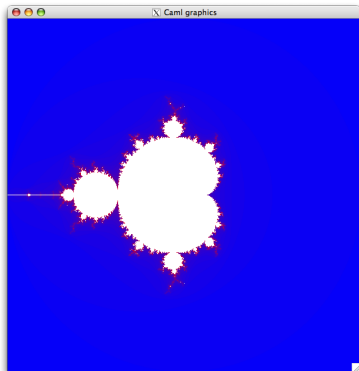
Étant donnée une séquence d'expressions  $e_1; e_2; \dots; e_n$

- ▶ toutes les expressions  $e_1, \dots, e_{n-1}$  ne font que des **effets de bord**
- ▶ la **valeur** et le **type** de la séquence sont ceux de la dernière expression  $e_n$

# Programme 4 : La fractale de Mandelbrot

Notion introduite :

- fonctions locales



# L'ensemble de Mandelbrot

Ensemble des points  $(a, b)$  du plan pour lesquels aucune des deux suites récurrentes suivantes ne tend vers l'infini (en valeur absolue)

$$\begin{aligned}x_0 &= 0 \\y_0 &= 0 \\x_{n+1} &= x_n^2 - y_n^2 + a \\y_{n+1} &= 2x_n y_n + b\end{aligned}$$

- ▶ pas de méthode exacte pour déterminer cette condition
- ▶ on peut démontrer que l'une de ces suites tend vers l'infini dès que  $x_n^2 + y_n^2 > 4$
- ▶ les points sont dans le cercle de rayon 2 centré en  $(0, 0)$

On dessine une approximation : ensemble des points  $(a, b)$  pour lesquels  $x_n^2 + y_n^2 \leq 4$  pour les  $k$  premières valeurs de ces suites

# les fonctions locales

Une déclaration de fonction peut être **locale** à une expression et ou locale à la déclaration d'une autre fonction

Ainsi, le programme suivant :

```
let boucle n =  
  let rec blc_rec i =  
    Printf.printf "%d " i;  
    if i<n then blc_rec (i+1)  
  in  
    blc_rec 0;;  
  
let carre x = x * x in boucle (carre 3);;  
  
affiche 0 1 2 3 4 5 6 7 8 9
```

# Définitions récursives

Une fonction est **récursive** si elle fait appel à elle même dans sa propre définition

Par exemple, la fonction factorielle  $n!$  peut être définie de manière récursive, pour tout entier  $n$ , par les deux équations suivantes

$$0! = 1$$

$$n! = n \times (n - 1)!$$

# Fonctions récursives en OCAML

La définition d'une fonction récursive est introduite par l'adjonction du mot clé **rec** au mot clé **let**

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

# processus d'évaluation

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

fact 3

# processus d'évaluation

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

$3 \neq 0 \Rightarrow$   $\text{fact } 3$   
 $3 * \text{fact}(3-1)$



# processus d'évaluation

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

```
fact 3  
3 ≠ 0 ⇒ 3 * fact(3-1)  
2 ≠ 0 ⇒ 3 * 2 * fact(2-1)
```

# processus d'évaluation

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

```
fact 3  
3 ≠ 0 ⇒ 3 * fact(3-1)  
2 ≠ 0 ⇒ 3 * 2 * fact(2-1)  
1 ≠ 0 ⇒ 3 * 2 * 1 * fact(1-1)
```

# processus d'évaluation

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

```
fact 3  
3 ≠ 0 ⇒ 3 * fact(3-1)  
2 ≠ 0 ⇒ 3 * 2 * fact(2-1)  
1 ≠ 0 ⇒ 3 * 2 * 1 * fact(1-1)  
0 = 0 ⇒ 3 * 2 * 1 * 1
```

# processus d'évaluation

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

```
fact 3  
3 ≠ 0 ⇒ 3 * fact(3-1)  
2 ≠ 0 ⇒ 3 * 2 * fact(2-1)  
1 ≠ 0 ⇒ 3 * 2 * 1 * fact(1-1)  
0 = 0 ⇒ 3 * 2 * 1 * 1  
      ⇒ 3 * 2 * 1
```

# processus d'évaluation

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

```
fact 3  
3 ≠ 0 ⇒ 3 * fact(3-1)  
2 ≠ 0 ⇒ 3 * 2 * fact(2-1)  
1 ≠ 0 ⇒ 3 * 2 * 1 * fact(1-1)  
0 = 0 ⇒ 3 * 2 * 1 * 1  
      ⇒ 3 * 2 * 1  
      ⇒ 3 * 2
```

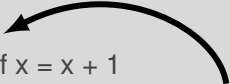
# processus d'évaluation

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

```
fact 3  
3 ≠ 0 ⇒ 3 * fact(3-1)  
2 ≠ 0 ⇒ 3 * 2 * fact(2-1)  
1 ≠ 0 ⇒ 3 * 2 * 1 * fact(1-1)  
0 = 0 ⇒ 3 * 2 * 1 * 1  
      ⇒ 3 * 2 * 1  
      ⇒ 3 * 2  
      ⇒ 6
```

# Pourquoi ce mot clé `rec` ?

La règle de portée du `let`



```
let f x = x + 1
```

```
let f y = if y=0 then f 2 else 4+y
```

# Pourquoi ce mot clé `rec` ?

La règle de portée du `let`

```
let f x = x + 1
```

```
let rec f y = if y=0 then f 2 else 4+y
```





## Que se passe-t-il si $n$ est négatif ?

1.  $n!$  est définie sur  $\mathbb{N}$
2. la fonction `fact` a pour type `int  $\rightarrow$  int` et **ne termine pas** si son paramètre est négatif

## Que se passe-t-il si $n$ est négatif ?

1.  $n!$  est définie sur  $\mathbb{N}$
2. la fonction `fact` a pour type `int  $\rightarrow$  int` et **ne termine pas** si son paramètre est négatif

Quelle **valeur** `fact` peut-elle rendre si  $n < 0$  ?

## Que se passe-t-il si $n$ est négatif ?

1.  $n!$  est définie sur  $\mathbb{N}$
2. la fonction `fact` a pour type `int  $\rightarrow$  int` et **ne termine pas** si son paramètre est négatif

Quelle **valeur** `fact` peut-elle rendre si  $n < 0$  ?

La seule solution raisonnable est de **stopper** le calcul en cours et d'**expliquer** le problème à l'utilisateur

On utilise pour cela la fonction prédéfinie **failwith** *m* qui termine un programme en affichant le message *m* à l'écran

## Implantation de fact (deuxième version)

```
let rec fact n =  
  if n<0 then failwith "argument negatif";  
  if n=0 then 1 else n * fact (n-1)  
;;
```

## Implantation de fact (deuxième version)

```
let rec fact n =  
  if n<0 then failwith "argument negatif";  
  if n=0 then 1 else n * fact (n-1)  
;;
```

Encore mieux, une fonction intermédiaire pour éviter des tests inutiles

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

```
let fact n =  
  if n<0 then failwith "argument negatif";  
  factorielle n  
;;
```

# Implantation de fact (troisième version)

On peut cacher la fonction "incorrecte" comme une fonction locale

```
let fact n =  
  let rec factorielle n =  
    if n=0 then 1 else n * factorielle (n-1)  
  in  
    if n<0 then failwith "argument negatif";  
    factorielle n  
;;
```

# Définitions par cas

Analyse par cas d'une valeur avec la construction `match-with`

```
let rec fact n =  
  match n with  
  | 0 -> 1  
  | _ -> n * fact (n-1)  
;;
```

Ceci permet de se rapprocher encore plus de la définition mathématique

**Réversivité double** : plusieurs appels récursifs peuvent apparaître dans la définition



**Récurtivité double** : plusieurs appels récursifs peuvent apparaître dans la définition

```
let rec fibonacci n =  
  match n with  
  | 0 -> 0  
  | 1 -> 1  
  | _ -> fibonacci (n-1) + fibonacci (n-2)  
;;
```

**Récurtivité mutuelle** : une fonction  $f$  peut faire référence à une fonction  $g$  qui elle-même fait référence à  $f$

Dans un fichier `pair.ml` :

```
let rec pair n = (n = 0) || impair (n-1) ;;
```

```
let rec impair n = (n <> 0) && pair (n-1) ;;
```

**Récurtivité mutuelle** : une fonction  $f$  peut faire référence à une fonction  $g$  qui elle-même fait référence à  $f$

Dans un fichier `pair.ml` :

```
let rec pair n = (n = 0) || impair (n-1) ;;
```

```
let rec impair n = (n <> 0) && pair (n-1) ;;
```

**compilation**

```
> ocamlc -o pair pair.ml
```

```
File "pair.ml", line 1, characters 28-34:
```

```
Unbound value impair
```

Pour définir deux fonctions  $f$  et  $g$  mutuellement récursives on utilise la construction **let rec  $f = e1$  and  $g = e2$**

```
let rec pair n = (n = 0) || impair (n-1)
and impair n = (n <> 0) && pair (n-1)
;;
```

# Principes d'une définition récursive

L'analyse par cas dans une définition récursive permet de regrouper

- ▶ les **cas de base** (**non rékursifs**)
- ▶ les cas **rékursifs** qui renvoient à la définition en cours

La définition d'une fonction récursive revient **toujours** à identifier ces différents cas puis à vérifier que :

# Principes d'une définition récursive

L'analyse par cas dans une définition récursive permet de regrouper

- ▶ les **cas de base** (**non rékursifs**)
- ▶ les cas **rékursifs** qui renvoient à la définition en cours

La définition d'une fonction récursive revient **toujours** à identifier ces différents cas puis à vérifier que :

1. les cas rékursifs finissent par renvoyer aux cas de base (afin de garantir la terminaison de la fonction)

# Principes d'une définition récursive

L'analyse par cas dans une définition récursive permet de regrouper

- ▶ les **cas de base** (**non rékursifs**)
- ▶ les cas **rékursifs** qui renvoient à la définition en cours

La définition d'une fonction récursive revient **toujours** à identifier ces différents cas puis à vérifier que :

1. les cas rékursifs finissent par renvoyer aux cas de base (afin de garantir la terminaison de la fonction)
2. tous les cas sont couverts dans la définition

# Principes d'une définition récursive

L'analyse par cas dans une définition récursive permet de regrouper

- ▶ les **cas de base** (**non rékursifs**)
- ▶ les cas **rékursifs** qui renvoient à la définition en cours

La définition d'une fonction récursive revient **toujours** à identifier ces différents cas puis à vérifier que :

1. les cas rékursifs finissent par renvoyer aux cas de base (afin de garantir la terminaison de la fonction)
2. tous les cas sont couverts dans la définition
3. on ne risque pas de boucler sur des cas rékursifs, en s'assurant par exemple que chaque appel rékursif renvoie à un cas *plus simple*



# Exemples de mauvaises définitions récursives

Problème de **terminaison** : les cas récursifs ne renvoient pas aux cas de base

```
let rec fibonacci n =  
  match n with  
  | 0 -> 0  
  | _ -> fibonacci (n-1) + fibonacci (n-2)  
;;
```

# Exemples de mauvaises définitions récursives

Problème de **terminaison** : les cas récursifs ne renvoient pas aux cas de base

```
let rec fibonacci n =  
  match n with  
  | 0 -> 0  
  | _ -> fibonacci (n-1) + fibonacci (n-2)  
;;
```

Problème de **couverture** des cas possibles

```
let mystere x =  
  let rec mystere x n =  
    match n with  
    | 0 -> 1  
    | 1 -> x * mystere (x/2) ((x mod 2)+n)  
  in  
  if x<=0 then 1 else mystere x 1 ;;
```

# Récursion infinie

Il arrive cependant parfois de vouloir écrire de "mauvaises" définitions récursives

Soit la fonction `zero` qui recherche (éventuellement indéfiniment) une valeur pour laquelle une fonction `f` s'annule

```
let zero f =  
  let rec recherche i =  
    if f i = 0 then i else recherche (i+1)  
  in  
    recherche 0  
;;
```

# Récursion terminale

# Exemple introductif

Dans un fichier `somme.ml` :

```
let rec somme n =  
  match n with  
  | 0. -> 0.  
  | _ -> n +. somme (n -. 1.) ;;  
  
print_float (somme 90000.) ;;
```

# Exemple introductif

Dans un fichier `somme.ml` :

```
let rec somme n =  
  match n with  
  | 0. -> 0.  
  | _ -> n +. somme (n -. 1.) ;;  
  
print_float (somme 90000.) ;;
```

compilation

```
ocamlc -o somme somme.ml
```

# Exemple introductif

Dans un fichier `somme.ml` :

```
let rec somme n =  
  match n with  
  | 0. -> 0.  
  | _ -> n +. somme (n -. 1.) ;;  
  
print_float (somme 90000.) ;;
```

## compilation

```
ocamlc -o somme somme.ml
```

## exécution

```
./somme
```

```
Fatal error: exception Stack_overflow
```

# Appels en attente

L'exécution du programme précédent devrait correspondre au processus d'évaluation suivant

```
somme 90000.  
⇒ 90000. +. somme 89999.  
⇒ 90000. +. 89999. +. somme 89998.  
⇒ ...  
⇒ 90000. +. 89999. +. 4049865001.  
⇒ 90000. +. 4049955000.  
⇒ 4050045000.
```

L'appel à somme  $n$  est en **attente** du résultat de l'appel à somme  $(n-1)$



# Appels en attente

L'exécution du programme précédent devrait correspondre au processus d'évaluation suivant

```
    somme 90000.  
⇒ 90000. +. somme 89999.  
⇒ 90000. +. 89999. +. somme 89998.  
⇒ ...  
⇒ 90000. +. 89999. +. 4049865001.  
⇒ 90000. +. 4049955000.  
⇒ 4050045000.
```

L'appel à `somme n` est en **attente** du résultat de l'appel à `somme (n-1)`

Malheureusement, quand le nombre d'appels en attente est trop grand le programme s'arrête !

# Pile d'appels

Pour exécuter les appels de fonctions, quelque soit le langage et le compilateur, on utilise une **pile d'appel**, dans laquelle on stocke (entre autre) les valeurs des **arguments** et l'**adresse de retour** de l'appel

La taille de cette pile croît donc en fonction du nombre d'appels en attente et elle "**déborde**" quand il y a trop d'appels en attente (message *Stack overflow*)

Exemple. sous un Linux standard la taille de la pile est fixée à 8Ko

# Appels terminaux

Dans une fonction  $f$ , un appel à une fonction  $g$  est **terminal** si le résultat de cet appel est le résultat de  $f$

Exemples :

```
let f x = g x ;;
```

# Appels terminaux

Dans une fonction  $f$ , un appel à une fonction  $g$  est **terminal** si le résultat de cet appel est le résultat de  $f$

Exemples :

```
let f x = g x ;;
```

```
let f x = if ... then g x else ... ;;
```

# Appels terminaux

Dans une fonction  $f$ , un appel à une fonction  $g$  est **terminal** si le résultat de cet appel est le résultat de  $f$

Exemples :

```
let f x = g x ;;
```

```
let f x = if ... then g x else ... ;;
```

```
let f x = if ... then ... else g x ;;
```

# Appels terminaux

Dans une fonction  $f$ , un appel à une fonction  $g$  est **terminal** si le résultat de cet appel est le résultat de  $f$

Exemples :

```
let f x = g x ;;
```

```
let f x = if ... then g x else ... ;;
```

```
let f x = if ... then ... else g x ;;
```

```
let f x = let y = ... in g y ;;
```

# Appels terminaux

Dans une fonction  $f$ , un appel à une fonction  $g$  est **terminal** si le résultat de cet appel est le résultat de  $f$

Exemples :

```
let f x = g x ;;
```

```
let f x = if ... then g x else ... ;;
```

```
let f x = if ... then ... else g x ;;
```

```
let f x = let y = ... in g y ;;
```

```
let f x = match x with ... | p -> g x | ... ;;
```

# Appels terminaux

Dans une fonction  $f$ , un appel à une fonction  $g$  est **terminal** si le résultat de cet appel est le résultat de  $f$

Exemples :

```
let f x = g x ;;
```

```
let f x = if ... then g x else ... ;;
```

```
let f x = if ... then ... else g x ;;
```

```
let f x = let y = ... in g y ;;
```

```
let f x = match x with ... | p -> g x | ... ;;
```

Une fonction est **récursive terminale** si ses appels récursifs sont tous terminaux



# Exécution des appels terminaux

Voici une version de somme **récursive terminale**

```
let rec somme_term acc n =  
  match n with  
  | 0. -> acc  
  | _ -> somme_term (n +. acc) (n -. 1.)  
;;  
let somme n = somme_term 0. n ;;
```

# Exécution des appels terminaux

Voici une version de somme **récursive terminale**

```
let rec somme_term acc n =  
  match n with  
  | 0. -> acc  
  | _ -> somme_term (n +. acc) (n -. 1.)  
;;  
let somme n = somme_term 0. n ;;
```

- ▶ Avec cette version **plus de débordement de pile**
- ▶ Le compilateur `ocamlc` traite de manière spéciale les appels terminaux : il remplace dans la pile d'appel la place occupée par `somme_term acc n` par l'appel `somme_term (n +. acc) (n -. 1.)`

# Récursion efficace

## Programmer avec des accumulateurs

- ▶ Principe analogue à l'ajout de variables auxiliaires en programmation impérative
- ▶ Ajout de fonctions auxiliaires avec des paramètres supplémentaires, appelés **accumulateurs**

# Récursion efficace

## Programmer avec des accumulateurs

- ▶ Principe analogue à l'ajout de variables auxiliaires en programmation impérative
- ▶ Ajout de fonctions auxiliaires avec des paramètres supplémentaires, appelés **accumulateurs**

Autre exemple, la fonction factorielle en version récursive terminale

```
let rec fact_term acc n =  
  match n with  
  | 0 -> acc  
  | _ -> fact_term (n*acc) (n-1)  
;;  
let fact n = fact_term 1 n ;;
```