Introduction à la programmation fonctionnelle

Notes de cours

Cours 1

17 janvier 2020

Sylvain Conchon

Calendrier

MCC : 2 TP notés + 1 Examen

- 17 janvier : Cours
- 24 janvier : Cours
- 31 janvier : TP
- 7 février : TP
- 14 février : TP
- 28 février : Cours
- 6 mars : TP
- 20 mars : **TP noté**
- 27 mars : Cours
- 3 avril : Soutien
- 10 avril : TP
- 24 avril : **TP noté**

`http://www.lri.fr/~conchon/IPF/`

Qu’est-ce que la programmation fonctionnelle ?

C’est un style de programmation dans lequel :

Qu’est-ce que la programmation fonctionnelle ?

C’est un style de programmation dans lequel :

Qu’est-ce que la programmation fonctionnelle ?

C’est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

Qu'est-ce que la programmation fonctionnelle ?

C'est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

  Q : Peut-on vraiment écrire des programmes sans boucles ?

Qu'est-ce que la programmation fonctionnelle ?

C'est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

  Q : Peut-on vraiment écrire des programmes sans boucles ?

- On passe des fonctions en arguments à d'autres fonctions

Qu'est-ce que la programmation fonctionnelle ?

C'est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

  Q : Peut-on vraiment écrire des programmes sans boucles ?

- On passe des fonctions en arguments à d'autres fonctions

  Q : À quoi ça peut bien servir ?

Qu'est-ce que la programmation fonctionnelle ?

C'est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

  Q : Peut-on vraiment écrire des programmes sans boucles ?
- On passe des fonctions en arguments à d'autres fonctions

  Q : À quoi ça peut bien servir ?
- On peut créer des fonctions pendant l'exécution du programme

Qu'est-ce que la programmation fonctionnelle ?

C'est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

  Q : Peut-on vraiment écrire des programmes sans boucles ?

- On passe des fonctions en arguments à d'autres fonctions

  Q : À quoi ça peut bien servir ?

- On peut créer des fonctions pendant l'exécution du programme

  Q : Comment ça marche ?

Qu'est-ce que la programmation fonctionnelle ?

C'est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

  Q : Peut-on vraiment écrire des programmes sans boucles ?
- On passe des fonctions en arguments à d'autres fonctions

  Q : À quoi ça peut bien servir ?
- On peut créer des fonctions pendant l'exécution du programme

  Q : Comment ça marche ?
- On peut définir des fonctions anonymes

Qu'est-ce que la programmation fonctionnelle ?

C'est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

  Q : Peut-on vraiment écrire des programmes sans boucles ?

- On passe des fonctions en arguments à d'autres fonctions

  Q : À quoi ça peut bien servir ?

- On peut créer des fonctions pendant l'exécution du programme

  Q : Comment ça marche ?

- On peut définir des fonctions anonymes

  Q : Comment appeler ces fonctions dans un programme ?

Qu'est-ce que la programmation fonctionnelle ?

C'est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

  Q : Peut-on vraiment écrire des programmes sans boucles ?
- On passe des fonctions en arguments à d'autres fonctions

  Q : À quoi ça peut bien servir ?
- On peut créer des fonctions pendant l'exécution du programme

  Q : Comment ça marche ?
- On peut définir des fonctions anonymes

  Q : Comment appeler ces fonctions dans un programme ?
- Les fonctions peuvent être polymorphes, c'est-à-dire s'appliquent sans distinction à tous les types de données

Qu'est-ce que la programmation fonctionnelle ?

C'est un style de programmation dans lequel :

- On écrit des fonctions récursives plutôt que boucles

  Q : Peut-on vraiment écrire des programmes sans boucles ?

- On passe des fonctions en arguments à d'autres fonctions

  Q : À quoi ça peut bien servir ?

- On peut créer des fonctions pendant l'exécution du programme

  Q : Comment ça marche ?

- On peut définir des fonctions anonymes

  Q : Comment appeler ces fonctions dans un programme ?

- Les fonctions peuvent être polymorphes, c'est-à-dire s'appliquent sans distinction à tous les types de données

  Q : Comment écrit-on le type d'une telle fonction ?

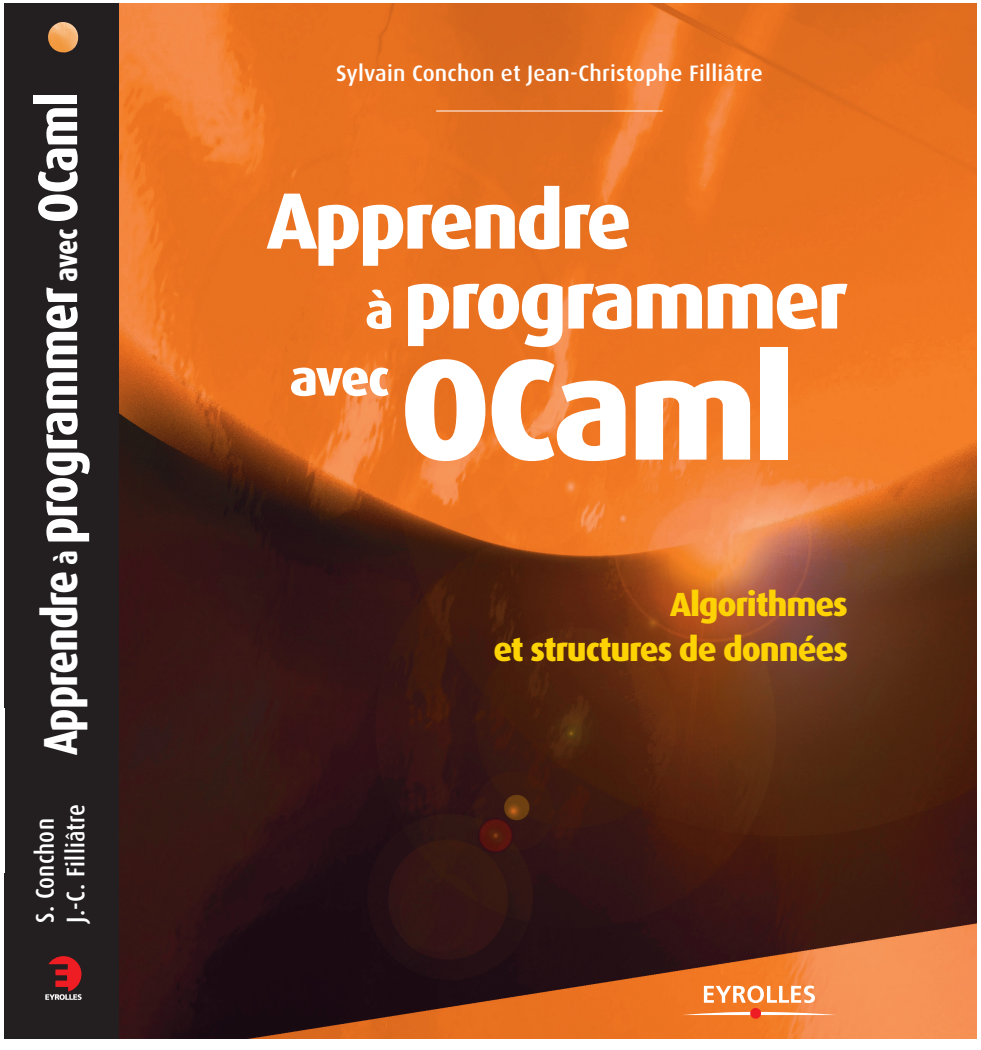
Structure du cours

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

Le langage OCaml

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.)

Les outils

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` exp_1 `then` exp_2 `else` exp_3

  l'expression exp_1 doit être de type `bool`.

Types et expressions élémentaires

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^e 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 connues 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

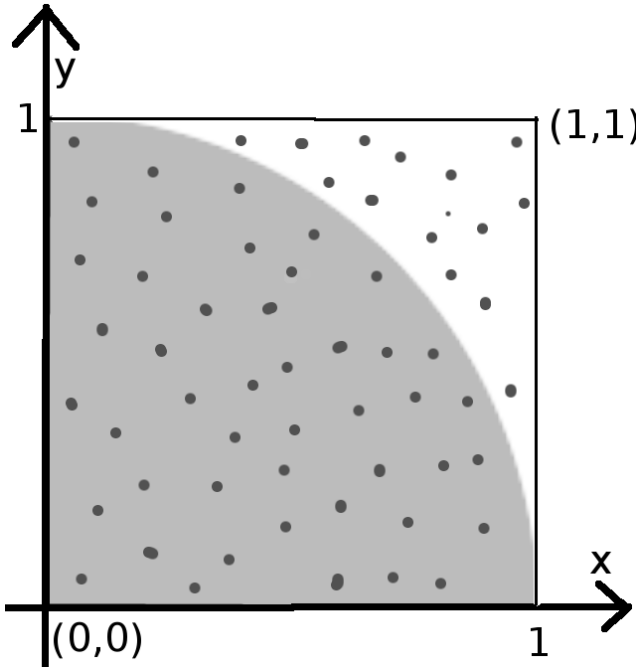
Programme 2 : Méthode de Monte-Carlo

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 π 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 $\mathtt{n}$ de points dans le carré, si $\mathtt{p}$ est le nombre de points à l'intérieur du cercle, alors $4 \times \mathtt{p}/\mathtt{n}$ donne une bonne approximation de π



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
```

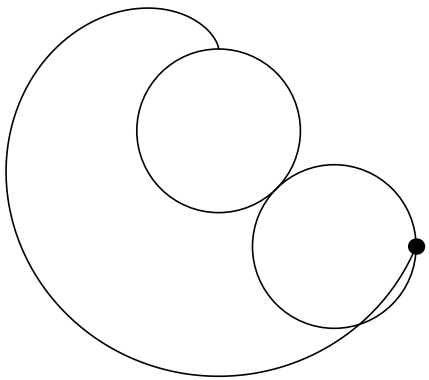
- l'ajout du mot-clé `rec` change la portée de l'identificateur : il est alors accessible dans la définition de la fonction

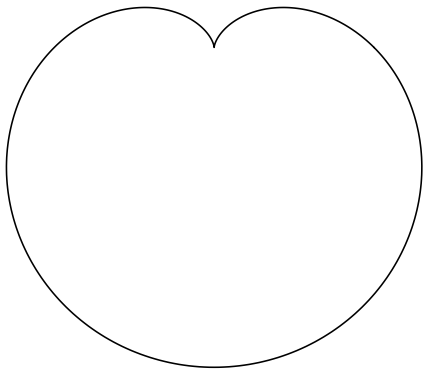
Programme 3 : Dessin d'une cardioïde

Notions introduites :

- bibliothèque `Graphics`
- Séquence d'expressions

Définition d'une cardioïde

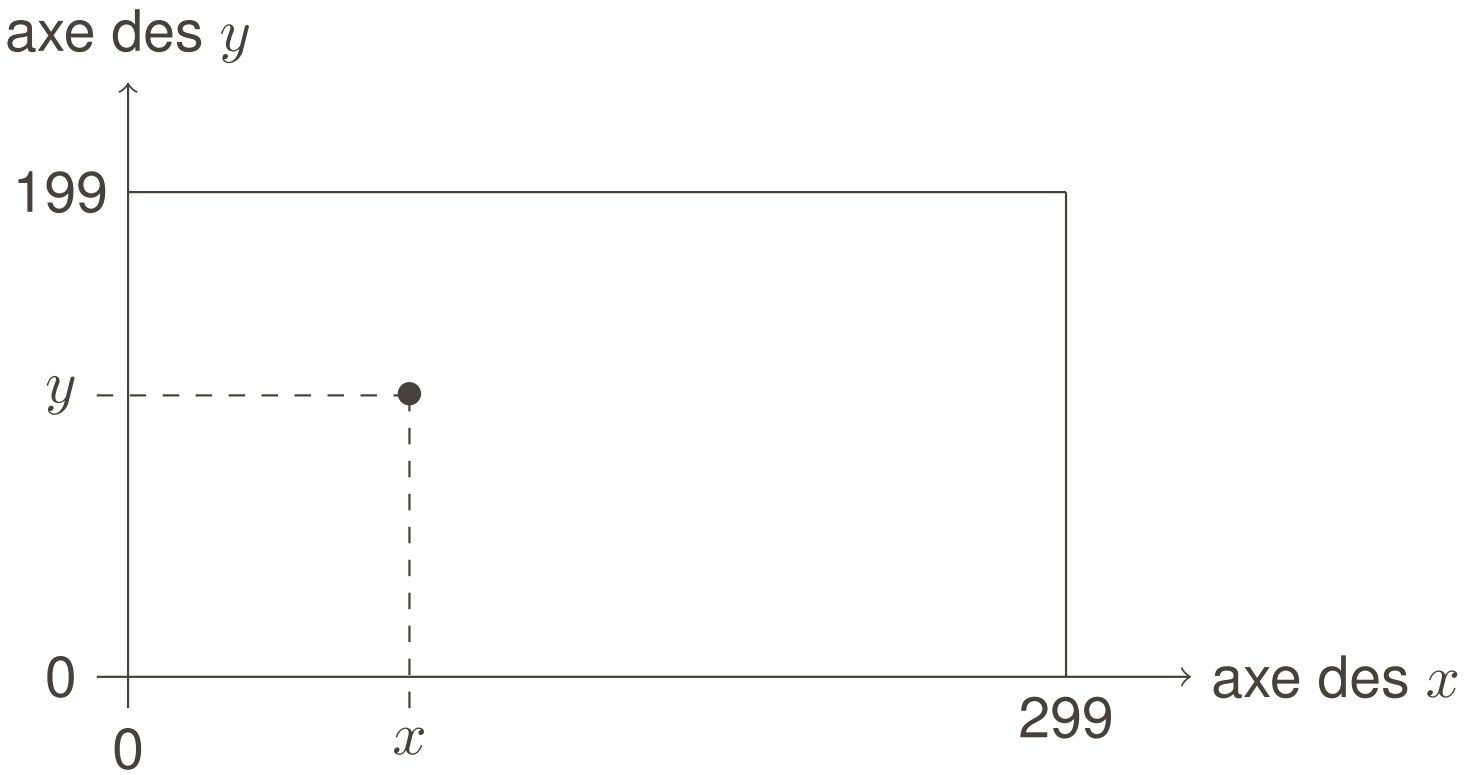




$$\begin{cases} x(\theta) = a\,(1 - sin(\theta))\,cos(\theta) \\ y(\theta) = a\,(1 - sin(\theta))\,sin(\theta) \end{cases}$$

La bibliothèque `Graphics`

- ajouter le fichier `graphics.cmxa` dans la commande de compilation (`ocamlopt ... graphics.cmxa ...`)
- 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



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`

D’autres bibliothèques graphiques

- LablTk : bibliothèque Tcl/Tk fournie avec OCaml, qui permet de concevoir des interfaces graphiques (GUI) avec menus déroulants, boutons, etc.
- LablGtk : similaire à LablTk mais basée sur la bibliothèque Gtk. Elle n’est pas distribuée avec OCaml ; on peut la télécharger à l’adresse

  `http://lablgtk.forge.ocamlcore.org/`
- OcamlSDL : bibliothèque SDL (*Simple DirectMedia Layer*) pour OCaml utilisée principalement pour réaliser des jeux vidéos. Elle est disponible à l’adresse

  `http://ocamlsdl.sourceforge.net/`

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

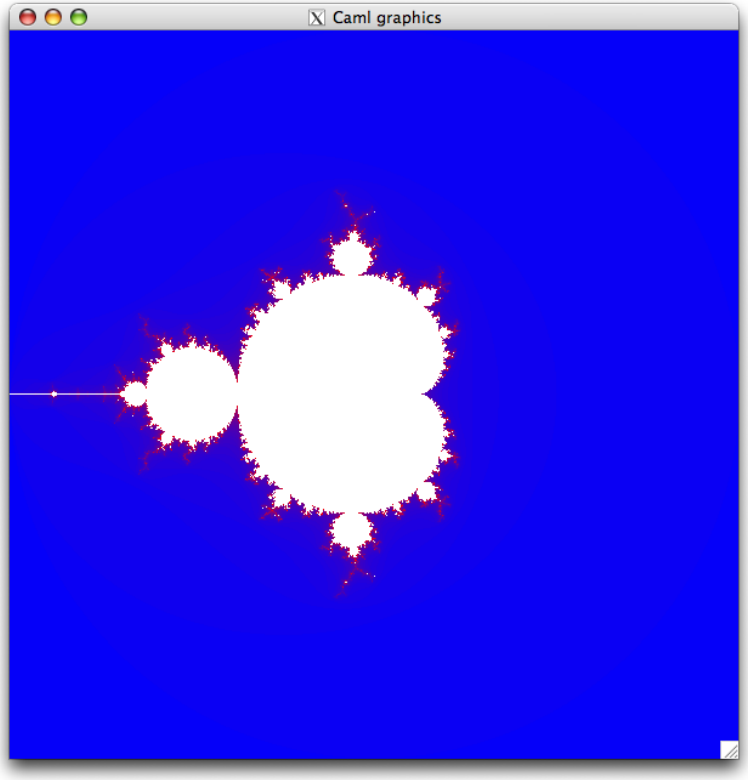
Étant donnée une séquence d'expressions $\mathtt{e}_1\,;\mathtt{e}_2\,;\cdots\,;\mathtt{e}_n$

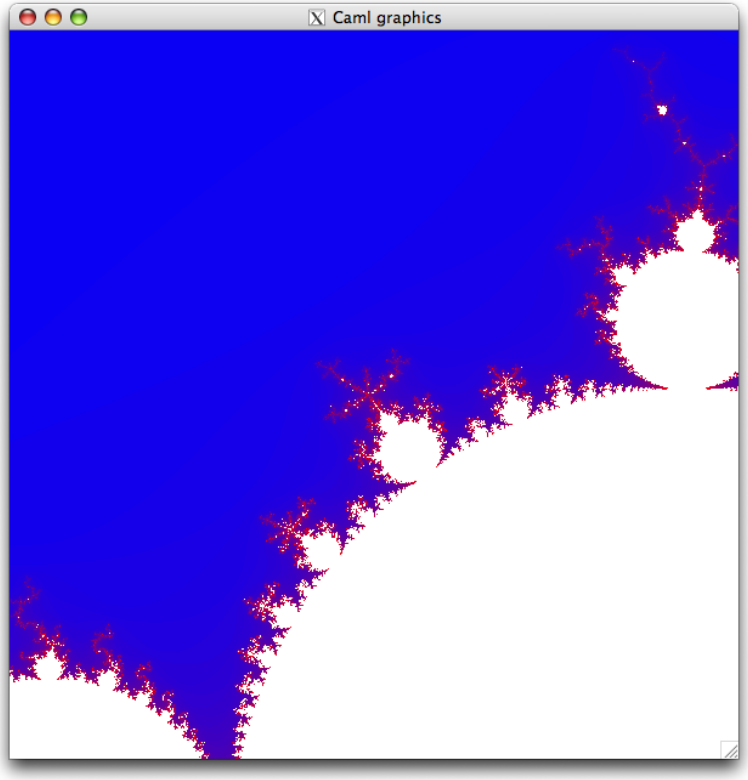
- toutes les expressions $\mathtt{e}_1, \ldots, \mathtt{e}_{n-1}$ ne font que des effets de bord
- la valeur et le type de la séquence sont ceux de la dernière expression $\mathtt{e}_n$

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`