λ

Programmation Fonctionnelle: Introduction

Yann Régis-Gianas
(IRIF, Univ. Paris Diderot, Inria) – yrg@irif.fr

2019-09-13

Plan

Qu'est-ce que la programmation?

Qu'est-ce que la programmation?

Plus que le "codage" vers la machine:
un acte de **communication** entre développeurs.

Qu'est-ce que la programmation?

Plus que le "codage" vers la machine:
un acte de **communication** entre développeurs.

Pour l'analyse des problèmes et leur implémentation.
Pour toutes les étapes de la vie d'un logiciel.
Pour bien faire le bon logiciel.

Les deux approches de la programmation que vous connaissez

Approche/paradigme de programmation

Ensemble de techniques d'analyse et d'implémentation induit par des mécanismes linguistiques. À chaque langage, une façon de penser.

Programmation procédurale

Mécanisme = lecture/écriture de données en mémoire.
Programme = modifier la mémoire jusqu'à l'obtention du résultat voulu.
Exemple: le tri en place.

Programmation orientée objet

Mécanisme = l'échange de messages entre objets.
Programme = faire collaborer des objets pour résoudre un problème.
Exemple: le progiciel qui implémente la logique métier de DupontCorp.

Les deux approches de la programmation que vous connaissez

Approche/paradigme de programmation

Ensemble de techniques d'analyse et d'implémentation induit par des mécanismes linguistiques. À chaque langage, une façon de penser.

Programmation procédurale

Mécanisme = lecture/écriture de données en mémoire.
Programme = modifier la mémoire jusqu'à l'obtention du résultat voulu.
Exemple: le tri en place.

Programmation orientée objet

Mécanisme = l'échange de messages entre objets.
Programme = faire collaborer des objets pour résoudre un problème.
Exemple: le progiciel qui implémente la logique métier de DupontCorp.

Qu'est-ce que l'approche fonctionnelle de la programmation?
La troisième grande approche de la programmation aujourd'hui.

Pour les mécanismes : retour aux fondamentaux!

Oubliez les variables!
Oubliez les pointeurs!
Oubliez la liaison tardive des appels de méthode!

Pour les mécanismes : retour aux fondamentaux!

Oubliez les variables!
Oubliez les pointeurs!
Oubliez la liaison tardive des appels de méthode!

Rappelez-vous des **règles de calculs** appris à l'école primaire!
Rappelez-vous des **simplifications d'équations** faites au collège!
Rappelez-vous de la notion de **fonction** étudiée au lycée!

Pour les mécanismes : retour aux fondamentaux!

Oubliez les variables!
Oubliez les pointeurs!
Oubliez la liaison tardive des appels de méthode!

Rappelez-vous des **règles de calculs** appris à l'école primaire!
Rappelez-vous des **simplifications d'équations** faites au collège!
Rappelez-vous de la notion de **fonction** étudiée au lycée!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

Pour les mécanismes : retour aux fondamentaux!

Oubliez les variables!
Oubliez les pointeurs!
Oubliez la liaison tardive des appels de méthode!

Rappelez-vous des **règles de calculs** appris à l'école primaire!
Rappelez-vous des **simplifications d'équations** faites au collège!
Rappelez-vous de la notion de **fonction** étudiée au lycée!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

Calculons "tri (1 ; 3 ; 2 ; 5)"! On procède par **remplacement d'expressions**!

Pour les mécanismes : retour aux fondamentaux!

Oubliez les variables!
Oubliez les pointeurs!
Oubliez la liaison tardive des appels de méthode!

Rappelez-vous des **règles de calculs** appris à l'école primaire!
Rappelez-vous des **simplifications d'équations** faites au collège!
Rappelez-vous de la notion de **fonction** étudiée au lycée!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

Calculons "tri (1 ; 3 ; 2 ; 5)"! On procède par **remplacement d'expressions**!
Aucune mémoire, pile ou compteur d'instructions n'entrent en jeu!

Comparez!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

```
1  Quicksort(A, low, high) {
2      if (low < high) {
3          pivot_location = Partition(A,low,high)
4          Quicksort(A,low, pivot_location-1)
5          Quicksort(A, pivot_location + 1, high)
6      }
7  }
8  Partition(A, low, high) {
9       pivot = A[low]
10      leftwall = low
11      for i = low + 1 to high {
12          if (A[i] < pivot) then{
13              swap(A[i], A[leftwall + 1])
14              leftwall = leftwall + 1
15          }
16      }
17      swap(pivot, A[leftwall])
18      return (leftwall)
19 }
```

Comparez!

tri : liste $\longrightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par (< x)) puis x, puis tri (l filtrée par ($\geq$ x))

- "tri" ne modifie pas la liste, elle **construit** une autre liste.

Comparez!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

- "tri" ne modifie pas la liste, elle **construit** une autre liste.

$\Rightarrow$ Le programme procédural **modifie** un tableau en place.

Comparez!

tri : liste $\longrightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

- "tri" ne modifie pas la liste, elle **construit** une autre liste.
- "tri" est défini par **induction** sur listes.

Comparez!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

- "tri" ne modifie pas la liste, elle **construit** une autre liste.

- "tri" est défini par **induction** sur listes.

$\Rightarrow$ Le programme procédural utilise des **indices**, et des **boucles**.

Comparez!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par (< x)) puis x, puis tri (l filtrée par ($\geq$ x))

- "tri" ne modifie pas la liste, elle **construit** une autre liste.
- "tri" est défini par **induction** sur listes.
- "tri" est défini par des équations avec à gauche une **expression**.

Comparez!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

- "tri" ne modifie pas la liste, elle **construit** une autre liste.
- "tri" est défini par **induction** sur listes.
- "tri" est défini par des équations avec à gauche une **expression**.

$\Rightarrow$ Le programme procédural modifie la machine via des **commandes**.

Comparez!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par (< x)) puis x, puis tri (l filtrée par ($\geq$ x))

- "tri" ne modifie pas la liste, elle **construit** une autre liste.
- "tri" est défini par **induction** sur listes.
- "tri" est défini par des équations avec à gauche une **expression**.
- "tri" s'appuie sur l'opérateur "l filtrée par p" où p est une **fonction**.

Comparez!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

- "tri" ne modifie pas la liste, elle **construit** une autre liste.
- "tri" est défini par **induction** sur listes.
- "tri" est défini par des équations avec à gauche une **expression**.
- "tri" s'appuie sur l'opérateur "l filtrée par p" où p est une **fonction**.

$\Rightarrow$ Le programme procédural ne peut définir de tels opérateurs.
On doit implémenter le filtrage à la main à chaque fois.

Comparez!

tri : liste $\rightarrow$ liste
tri (liste vide) = liste vide
tri (x puis l) = tri (l filtrée par ($< x$)) puis x, puis tri (l filtrée par ($\geq x$))

- "tri" ne modifie pas la liste, elle **construit** une autre liste.
- "tri" est défini par **induction** sur listes.
- "tri" est défini par des équations avec à gauche une **expression**.
- "tri" s'appuie sur l'opérateur "l filtrée par p" où p est une **fonction**.
- "tri" est évidemment correcte!

Est-ce vraiment possible?

Est-ce vraiment possible?

En OCaml:

```
let rec sort = function
| [] ->
  []
| x :: xs ->
  sort (filter (( < ) x) xs) @ [x] @ sort (filter (( >= ) x) xs)
```

En Haskell:

```
sort [] =
  []
sort (x : xs) =
  sort [a | a <- xs, a <= x] ++ [x] ++ sort [a | a <- xs, a > x]
```

L'approche fonctionnelle de la programmation

Une programmation qui prend les fonctions aux sérieux!
– Xavier Leroy

L'approche fonctionnelle de la programmation

Une programmation qui prend les fonctions aux sérieux!
– Xavier Leroy

Programmation fonctionnelle = pureté + induction + fonction

La pureté

Voici un programme:

```
1  y = f (42);
2  z = f (42);
```

En général, dans un langage impératif – i.e. où le calcul avance par modification de la mémoire – peut-on affirmer que `y = z`?

La pureté

Voici un programme:

```
1  y = f (42);
2  z = f (42);
```

En général, dans un langage impératif – i.e. où le calcul avance par modification de la mémoire – peut-on affirmer que `y = z`? **NON**.

La pureté

Voici un programme:

```
1  y = f (42);
2  z = f (42);
```

En général, dans un langage impératif – i.e. où le calcul avance par modification de la mémoire – peut-on affirmer que `y = z`? **NON**.

Définition

Une fonction est **pure** si son évaluation ne modifie pas son environnement. Elle associe toujours la même valeur pour une valeur d'entrée donnée.

Si `f` est pure, `y = z`! Et d'ailleurs, cela ne sert probablement à rien de calculer `f (42)` deux fois!

L'évaluation d'un programme fonctionnel produit de nouvelles données sans détruire les données déjà calculées (sauf si elles ne sont plus utiles, bien sûr).

L'induction

Pourquoi la fonction "tri" est-elle "évidemment correcte"?

L'induction

Pourquoi la fonction "tri" est-elle "évidemment correcte"?

- Parce qu'elle traite toutes les formes possibles de liste ; et
- parce qu'elle se rappelle sur des listes toujours plus petites ;

L'induction

Pourquoi la fonction "tri" est-elle "évidemment correcte"?

- Parce qu'elle traite toutes les formes possibles de liste ; et
- parce qu'elle se rappelle sur des listes toujours plus petites ;

nous pouvons appliquer le principe de **raisonnement par induction**:

- La liste vide triée contient bien les éléments du vide par ordre croissant.
- Notons $l_{<x}$ pour l'évaluation de "l filtrée par $(< x)$", $l_{\geq x}$ pour celle de "l filtrée par $(\geq x)$)".
  Par induction, "tri $(l_{<x})$" contient les éléments de $l_{<x}$ triés et comme ils sont tous plus petits que x : "tri $l_{<x}$ suivi de x" est bien triée. Par un raisonnement symétrique, "tri $l_{<x}$ suivi de x suivi de $l_{\geq x}$" est aussi triée. Comme $l_{<x}$, x et $l_{\geq x}$ forment un partitionnement de l, ce sont bien les éléments de l que l'on vient de trier.

Conclusion: Quelque soit la liste l donnée à "tri", tri(l) contient exactement les éléments de l triés par ordre croissant. Qed.

L'induction n'est pas un problème mais une solution!

L'induction n'est pas un problème mais une solution!

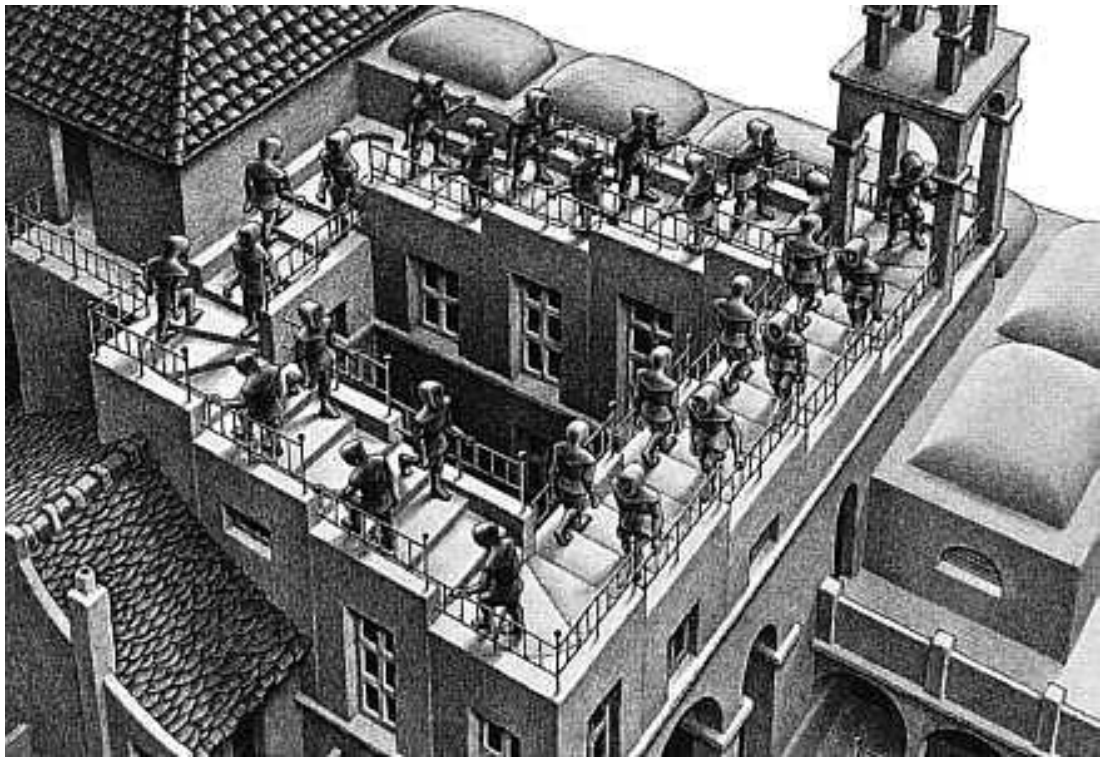
Pourquoi Roméo n'a-t-il pas utilisé les escaliers?

L'induction n'est pas un problème mais une solution!

Pourquoi Roméo n'a-t-il pas utilisé les escaliers?
Parce qu'il ne connaissait pas l'induction.

Peu importe le nombre de marches, tout escalier est fiable

À condition ce soit bien un escalier, bien entendu!



Des répétitions liées à une faiblesse d'expressivité

Après avoir programmé suffisamment longtemps en C, on écrit:

```
1  for (int i = 0; i < N; i++) {
2    ...
3  }
```

sans même sans rendre compte! D'autres motifs plus sophistiqués sont presqu'aussi courant. C'est le cas de l'accumulation:

```
1  accu = ... // initialisation
2  for (int i = 0; i < N; i++) {
3     ...
4     accu = ...; // mise à jour de accu
5     ...
6  }
```

Que de caractères répétés! Mais peut-on faire autrement en C?

Les fonctions comme valeurs : l'ordre supérieur

Il faudrait pouvoir dire que les `...` sont des paramètres d'une fonction "accumulate" définie comme suit dans un langage C imaginaire:

```
1  accu_t accumulate (int N, accu_t initialization, ??? update) {
2     accu_t accu = initialization;
3     for (int i = 0; i < N; i++) {
4        accu = update (i, accu);
5     }
6     return accu;
7  }
```

Ensuite, on écrirait quelque chose comme:

```
1  accu = accumulate (array_len, 0, { (i, accu) ->
2     accu = accu + i;
3  }); // Ici, on passe une fonction, comme toute autre valeur!
```

Nous aurions drastiquement factorisé notre code! Cette modularité est reservé aux langages d'ordre supérieur: fonctionnels et objets.
(Nous verrons que les pointeurs de fonction ne sont pas des fonctions!)

Des types pour structurer la programmation

Dans ce cours, nous nous intéresserons surtout à:

La programmation fonctionnelle **typée**

Définition

Un **type** est une information statique sur la nature d'une valeur.

Le type est donc un **outil de spécification** pour le programmeur.

Comme en Java, où on utilise le nom des classes et des interfaces pour représenter un concept. Les types nous servirons à associer des **propriétés aux valeurs**.

Plan

Les problèmes de la programmation

- Comment obtenir des programmes qui s'exécutent **sans erreur**?
- Quel langage choisir pour **maximiser sa liberté d'expression**?
- Comment développer des systèmes qui **passent à l'échelle**?
- Comment **affronter la complexité** des problèmes des utilisateurs?
- Comment **minimiser la consommation** par les programmes?

Programmer de façon sûre

Deux grands types d'erreur arrivent à l'exécution:

1. Le programme "plante"
   parce qu'il essaie de faire quelque chose qui n'a pas de sens.
2. Le programme calcule sans planter un résultat faux
   à cause d'une erreur dans la logique du programme.

Programmer de façon sûre

Deux grands types d'erreur arrivent à l'exécution:

1. Le programme "plante"
   parce qu'il essaie de faire quelque chose qui n'a pas de sens.
2. Le programme calcule sans planter un résultat faux
   à cause d'une erreur dans la logique du programme.

Les langages fonctionnels typés **garantissent** l'absence d'erreurs du type 1.

Programmer de façon sûre

Deux grands types d'erreur arrivent à l'exécution:

1. Le programme "plante"
   parce qu'il essaie de faire quelque chose qui n'a pas de sens.
2. Le programme calcule sans planter un résultat faux
   à cause d'une erreur dans la logique du programme.

Les langages fonctionnels typés **garantissent** l'absence d'erreurs du type 1.

Le **haut niveau d'abstraction** des programmes fonctionnels permet d'écrire du code **concis** et dont **le sens est clair**: cela permet de réduire les erreurs du type 2.

Programmer de façon sûre

Deux grands types d'erreur arrivent à l'exécution:

1. Le programme "plante"
   parce qu'il essaie de faire quelque chose qui n'a pas de sens.
2. Le programme calcule sans planter un résultat faux
   à cause d'une erreur dans la logique du programme.

Les langages fonctionnels typés **garantissent** l'absence d'erreurs du type 1.

Le **haut niveau d'abstraction** des programmes fonctionnels permet d'écrire du code **concis** et dont **le sens est clair**: cela permet de réduire les erreurs du type 2.

Mais surtout ...

Programmer le bon langage pour programmer

Les langages fonctionnels sont très **expressifs**.

- Informellement, l'expressivité d'un langage, c'est sa capacité à réduire la distance entre la façon dont on souhaite exprimer une solution et la façon dont on l'exprime effectivement.
- Depuis LISP[1], les langages fonctionnels offrent des mécanismes pour **programmer son propre langage**.
- On peut alors choisir le langage le mieux adapté à exprimer sa solution.

1

[1] Le tout premier langage fonctionnel, créé par McCarthy, le père de l'IA, en 1958.

Programmer pour passer à l'échelle

Comment construire :
des systèmes qui répondent à des millions d'utilisateurs?
des systèmes capables de traiter des téraoctets de données?

Programmer pour passer à l'échelle

Comment construire :
des systèmes qui répondent à des millions d'utilisateurs?
des systèmes capables de traiter des téraoctets de données?

Sans état, toute sous-expression peut s'évaluer en parallèle.

Programmer pour passer à l'échelle

Comment construire :
des systèmes qui répondent à des millions d'utilisateurs?
des systèmes capables de traiter des téraoctets de données?

Sans état, toute sous-expression peut s'évaluer en parallèle.

Dans l'industrie du logiciel

- MapReduce de Google
- Lambda chez Amazon AWS
- La programmation réactive

Proposer des solutions générales et recomposable

Une tension du développement logiciel:
Les besoins évoluent perpétuellement.
Comment adapter rapidement les logiciels?

- Approche orientée objet : rajouter une nouvelle sous-classe.
- Approche fonctionnelle : instancier différemment l'existant.

Extensibilité VS Généralité

Comment?

- Un programme fonctionnel =
  Composition de petits blocs généraux recomposables
- Un typage le plus général possible grâce au **polymorphisme paramétrique inféré automatiquement**.

Le typeur d'OCaml

```
# let rec sort = function
  | [] -> []
  | x :: xs -> sort (filter (( < ) x) xs) @ [x] @ sort (filter (( >= ) x) xs)
val sort : 'a list -> 'a list = <fun>
```

Le typeur d'OCaml a découvert la généralité de cette fonction.

Le typeur d'OCaml

```
# let rec sort = function
  | [] -> []
  | x :: xs -> sort (filter (( < ) x) xs) @ [x] @ sort (filter (( >= ) x) xs)
val sort : 'a list -> 'a list = <fun>
```

Le typeur d'OCaml a découvert la généralité de cette fonction.

On dit qu'il a **inféré le type** de la fonction `sort`.

Le typeur d'OCaml

```
# let rec sort = function
  | [] -> []
  | x :: xs -> sort (filter (( < ) x) xs) @ [x] @ sort (filter (( >= ) x) xs)
val sort : 'a list -> 'a list = <fun>
```

Le typeur d'OCaml a découvert la généralité de cette fonction.

On dit qu'il a **inféré le type** de la fonction `sort`.

Il peut révéler une généralité insoupçonnée dans nos programmes.

Réaliser des applications performantes

Comment minimiser les resources consommées par un logiciel?

Réaliser des applications performantes

Comment minimiser les resources consommées par un logiciel?

L'approche classique

- En optimisant ses parties critiques!
- Deux approches complémentaires, difficiles à concilier :
  1. Utiliser des algorithmes et structure de données sophistiqués.
  2. Optimiser le code machine de bas niveau.
- Les langages fonctionnels modernes offrent de bons compromis.
- Ils savent aussi donner la main à du code C/Rust.

Réaliser des applications performantes

Comment minimiser les resources consommées par un logiciel?

L'approche classique

- En optimisant ses parties critiques!
- Deux approches complémentaires, difficiles à concilier :
  1. Utiliser des algorithmes et structure de données sophistiqués.
  2. Optimiser le code machine de bas niveau.
- Les langages fonctionnels modernes offrent de bons compromis.
- Ils savent aussi donner la main à du code C/Rust.

Une idée plus profonde

- Réfléchir à une structuration globale différente des calculs.
- Exemples: la programmation par flux (Spark), incrémentale, réactive, ...
- Ce sont des **modèles de calcul alternatifs**.
- Les langages fonctionnels sont faits pour faciliter leur implémentation.

Plan

Les langages avec des traits fonctionnels

Voici quelques langages qui ont des traits fonctionnels:

- Java 8, C++, Python
- Clojure, LISP, Scheme, JavaScript, Erlang
- OCaml, ReasonML, Scala, Haskell, Coq

Lequel choisir?

Les langages avec des traits fonctionnels

Voici quelques langages qui ont des traits fonctionnels:

- Java 8, C++, Python
- Clojure, LISP, Scheme, JavaScript, Erlang
- OCaml, ReasonML, Scala, Haskell, Coq

Lequel choisir?

Seuls les langages de la troisième ligne ont les propriétés suivantes:

- Ils permettent la programmation inductive.
- Ils sont typés.
- Ils sont compilés.

(Les langages de la seconde ligne sont intéressants malgré tout!)

OCaml en deux mots

Propriétés

- Tous les mécanismes de la programmation fonctionnelle.
- Des mécanismes impératifs ($\neq$ Haskell ou Coq).
- Un système de types riches et un moteur d'inférence de type.
- Un système de module expressif.
- Un compilateur vers du code octet et du code natif.
- Un environnement d'exécution léger ($\neq$ Scala).

Un langage qui séduit

- OCaml est utilisé par MirageOS, une bibliothèque pour unikernel (cloud).
- OCaml est au cœur de plusieurs blockchains (e.g. Tezos).
- ReasonML [2] enthousiasme la communauté JS et Facebook.
- OCaml est traditionnellement utilisé pour écrire des compilateurs et analyseurs de code.

[2] (= OCaml avec une autre syntaxe)

Un tour rapide du langage OCaml

Nous utiliserons régulièrement des notebooks en ligne pour expliquer les constructions du langages:

`https://sketch.sh/s/agM8OEOPPCmcUOoO9GPWBa/`

Dans ce tour d'OCaml:

- Des expressions, plus de commandes!
- Des types pour ces expressions
- Des fonctions de première classe
- De l'analyse de motif (pattern-matching)
- Un système de modules

L'environnement de développement d'OCaml

`http://www.ocaml.org`

- OCaml est disponible dans la plupart des distributions Linux:

```
% sudo apt install ocaml
```

- La distribution standard d'OCaml fournit:
  1. un compilateur natif: **`ocamlopt`**
  2. un compilateur vers le code octet d'une machine virtuelle : **`ocamlc`**
  3. une boucle interative : **`ocaml`**
  4. des outils de profilage, de débogage, un Lex, un Yacc, ...
  5. une bibliothèque standard minimaliste.

L'écosystème OCaml

La communauté des programmeurs OCaml est très dynamique!

- Elle est formée de professionnels et d'amateurs qui collaborent pour construire un écosystème riche et de qualité.[3]
- Elle offre aux développeurs OCaml des outils très pratiques:
  - OPAM `https://opam.ocaml.org/`:
    le gestionnaire de paquets d'OCaml. 2400 paquets en 2019.
  - Dune `https://dune.build/`:
    le système pour automatiser la compilation des projets OCaml.
  - Merlin `https://github.com/ocaml/merlin`:
    un outil d'assistance à la programmation intégrée à votre éditeur.
  - Utop `https://github.com/ocaml-community/utop`:
    une boucle interactive ergonomique et sophistiquée.
- Elle répondra à vos questions sur IRC, Reddit, Discourse, StackOverflow.

[3] Vous voulez contribuez? Parlez m'en!

Citation

> Sometimes, the elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function.
> – John Carmack, creator of Doom, Quake, ...

Citation

Code is like humor. When you have to explain it, it's bad.
– Cory House

Citation

A programmer does not primarily write code; rather, he primarily writes to another programmer about his problem solution. The understanding of this fact is the final step in his maturation as technician.
– Anonymous

Citation

> The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.
> – Dijkstra

Citation

A programming language is low level when its programs require attention to the irrelevant.
– Alan Perlis

Citation

A language that doesn't affect the way you think about programming is not worth knowing.
– Alan Perlis

Citation