Programmation fonctionnelle et logique

INF6120

Samuele Giraudo

UQÀM

Automne 2026

1. Introduction

1.1. Informations générales

1.1.1. Informations pratiques

- Titre du cours : *Programmation fonctionnelle et logique*
- Sigle : INF6120
- Département : Informatique
- Coordinateurs : Samuele Giraudo
- Enseignant : Samuele Giraudo
- Courriel : giraudo.samuele@uqam.ca
- Bureau : PK-4430
- Page personnelle : https://www.giraudo.uqam.ca
- Site du cours : https://www.giraudo.uqam.ca/Teaching/INF6120/2026-09/INF6120.html
- Plan de cours : https://info.uqam.ca/plan_cours/Automne%202026/INF6120.html

Prérequis :

- validation d'INF3105, *Structures de données et algorithmes* ;

ainsi que toutes ses **dépendances** :

- validation d'INF1120, *Programmation I* ;
- validation d'INF1132, *Mathématiques pour l'informatique* **ou** de MAT1060, *Mathématiques algorithmiques* ;
- validation d'INF2120, *Programmation II*.

Calendrier des évaluations avec leurs pondérations :

1. séance 5 : examen 1, 33 % ;
2. séance 10 : examen 2, 33 % ;
3. séance 15 : examen 3, 34 %.

Contenu général :

- l'examen 1 porte sur ce qui a été vu dans les 4 premières séances de cours (programmation fonctionnelle) ;
- l'examen 2 porte sur ce qui a été vu dans les 8 premières séances de cours (programmation fonctionnelle) ;
- l'examen 3 porte sur ce qui a été vu dans toutes les 12 premières séances de cours (programmation fonctionnelle et programmation logique).

Pour les examens, aucun document n'est autorisé. Tout appareil électronique est interdit.

Pour valider le cours, il faut avoir une moyenne supérieure à 50 %.

1.1.2. Aperçu

Objectifs principaux :

- connaître les principes généraux du paradigme de programmation fonctionnel ;
- connaître les principes généraux du paradigme de programmation logique ;
- savoir développer des applications en OCaml ;
- savoir développer des applications en Prolog.

Ce n'est pas un cours d'apprentissage de langages, c'est un cours de découverte de paradigmes.

Le désavantage à cela est que la connaissance spécifique de ces deux langages sera relativement modérée.

L'avantage est que cela permet très facilement d'apprendre d'autres langages s'inscrivant dans des paradigmes similaires.

Le cours est organisé en deux parties de volumes respectifs environ 75 % / 25 %.

1. Programmation fonctionnelle.
   - Bases théoriques.
   - Machines de Turing et λ-calcul.
   - Caractéristiques principales des langages de programmation.
   - Programmation en OCaml.
   - Principes de programmation fonctionnelle.
   - Démonstration de justesse de programmes.
   - Stratégies d'évaluation.
2. Programmation logique.
   - Bases théoriques.
   - Unification et résolution.
   - Manipulation des listes.

1.1.3. Références

Les deux **références principales** suivantes couvrent une grande partie de ce cours.

1. M. R. Clarkson, OCaml Programming: Correct + Efficient + Beautiful, Fall 2026 Edition, 2026, https://cs3110.github.io/textbook/cover.html.
2. P. Flach, Simply Logical – Intelligent Reasoning by Example (Fully Interactive Online Edition), 2022, https://book.simply-logical.space/src/simply-logical.html.

Elles sont consultables librement aux liens indiqués.

Quelques **références secondaires** utiles et très intéressantes :

1. Y. Minsky, A. Madhavapeddy, Real World OCaml, 2nd Edition, 2022, https://dev.realworldocaml.org/.
2. X. Leroy *et al.*, The OCaml system release 5.5, 2026, https://ocaml.org/manual/5.5/index.html.

Quelques **autres références**, pour approfondir, mais non obligatoires :

1. G. Dowek, J.-J. Lévy, Introduction à la théorie des langages de programmation, École Polytechnique, 2006.
2. C. Okasaki, Purely Functional Data Structures, Cambridge University Press, 1999.
3. S. L. Peyton Jones, D. Lester, Implementing Functional Languages, Prentice Hall, 1992.
4. P. Hudak, D. Quick, The Haskell School of Music: From Signals to Symphonies, Cambridge University Press, 2018.

1.2. Introduction

1.2.1. Arbres binaires de recherche en programmation fonctionnelle

Un **arbre binaire** t est

1. soit une feuille :

$$t = \bullet$$

2. soit un nœud attaché à deux arbres binaires t_1 et t_2 :

$$t = \quad .$$

C'est une définition **récursive** car la définition de la notion fait référence à la notion qui est en cours de définition.

Les arbres binaires sont en général très faciles à implanter dans des langages fonctionnels.

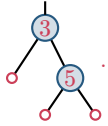
En OCaml, ceci peut se faire via la **définition du type**

```
type binary_trees =
    |Leaf
    |Node of binary_trees * int * binary_trees
```

L'expression

```
Node (Leaf, 3, Node (Leaf, 5, Leaf))
```

est de type `binary_trees` et désigne l'arbre binaire



Les entités `Leaf` et `Node` permettent de construire des données du type `binary_trees`. Ce sont des **constructeurs** de ce type.

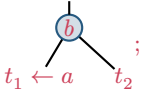
L'**algorithme d'insertion dans un arbre binaire de recherche** admet la description suivante.

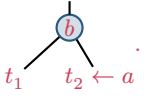
Soit t un arbre binaire de recherche et a une valeur.

L'arbre $t \leftarrow a$, obtenu en **insérant** a dans t, est défini par :

1. si t est une feuille, alors $t \leftarrow a :=$ 

;
2. sinon, t est de la forme $t =$ 

où b est une valeur. Dans ce cas :
   - si $a \leq b$, alors
   $$t \leftarrow a := \quad ;$$
   

   - sinon, $a > b$ et
   $$t \leftarrow a := \quad .$$
   

En OCaml, l'algorithme précédent peut s'implanter via la définition de fonction

```
let rec insert t a =
    match t with
        |Leaf -> Node (Leaf, a, Leaf)
        |Node (t1, b, t2) when a <= b -> Node (insert t1 a, b, t2)
        |Node (t1, b, t2) -> Node (t1, b, insert t2 a)
```

La construction d'un arbre binaire de recherche peut se faire par insertions successives :

```
let t0 = insert Leaf 3 in
let t1 = insert t0 4 in
let t2 = insert t1 2 in
let t3 = insert t2 3 in
let t4 = insert t3 4 in
let t5 = insert t4 1 in
t5
```

Toute cette expression s'évalue en une valeur de type `binary_trees`. Il s'agit d'un arbre binaire de recherche ayant 5 nœuds internes.

Nous verrons plus tard qu'il existe des manières beaucoup plus concises pour écrire ces sortes de définitions.

1.2.2. Instructions *vs* expressions

Considérons le problème qui, étant donnée une liste `lst` en entrée, consiste à calculer la sous-liste de `lst` obtenue en prenant un élément sur deux.

Sous le **paradigme impératif**, ceci peut se faire en Python via la fonction

```
def one_of_two(lst):
    res = []
    for i in range(len(lst)):
        if i % 2 == 0:
            res.append(lst[i])
    return res
```

Le calcul de `one_of_two([0, 1, 2, 3, 4])` utilise de la **mémoire** annexe

- pour construire le résultat `res` ;
- pour maintenir la variable `i` ;
- en interne, pour les fonctions appelées (`range`, `append`, *etc.*) ;
- pour enregistrer l'**état de l'exécution** (adresse de l'instruction exécutée).

Les variables `res` et `i` sont lues et **modifiées** au cours du temps.

Sous le paradigme fonctionnel, dans un OCaml de débutant, nous pouvons considérer la fonction

```
let rec one_of_two lst =
    if (List.length lst) <= 1 then
        lst
    else
        (List.hd lst) :: (one_of_two (List.tl (List.tl lst)))
```

Pour calculer l'**expression** `one_of_two [0; 1; 2; 3; 4]`, on l'**évalue** :

`one_of_two [0; 1; 2; 3; 4]` $\longrightarrow$ `0 :: one_of_two [2; 3; 4]` $\longrightarrow$ `0 :: 2 :: one_of_two [4]` $\longrightarrow$ `0 :: 2 :: [4]` $\Rightarrow$ `[0; 2; 4]`.

Dans ce cas présent, il n'y a

- ni utilisation externe de la mémoire ;
- ni état d'avancement du calcul qui dépend du temps.

Le calcul se fait en **réécrivant** l'expression tant que possible. Les définitions de **fonctions** permettent d'expliquer comment réaliser un calcul.

L'expression est en mémoire, mais il n'y a pas besoin d'enregistrer des données annexes.

1.2.3. Principes généraux de la programmation fonctionnelle

Quelques caractéristiques de la programmation fonctionnelle (en OCaml spécifiquement) :

- **Allocation** de mémoire **implicite**.

  Le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

  L'écriture d'un programme est rapide et est fidèle aux définitions des objets et algorithmes manipulés.

- Le **compilateur** est **exigeant** et vérifie beaucoup de choses.

  En pratique :

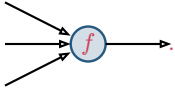
  *« Le compilateur OCaml n'accepte **presque jamais** mon code mais quand il l'accepte, ça marche **presque toujours** comme je le veux. »*

  vs

  *« Le compilateur X accepte **presque toujours** mon code mais ça ne marche **presque jamais** comme je le veux. »*

La **programmation fonctionnelle** est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions**.

Une fonction f est une entité qui accepte des valeurs en entrée et qui produit une valeur en sortie :



On **évite les effets secondaires** (et donc, en particulier, on ne fait pas d'affectation). Les fonctions ne font rien d'autre que de renvoyer des valeurs.

On **évite les séquences d'instructions impératives** (et donc, en particulier, on n'utilise pas de boucles `while`, `do while`, `for` ou autres).

On utilise en revanche (beaucoup) la **définition de fonctions**, l'**application de fonctions** et la **récursivité**.

1.2.4. Fonctions *vs* prédicats

Considérons le problème qui, étant donné une liste `lst` et une valeur `x` en entrées, consiste à connaître la position d'une occurrence de `x` dans `lst`.

Sous le **paradigme impératif**, ceci peut se faire en Python via la fonction

```python
def first(lst, x):
    i = 0
    for i in range(len(lst)):
        if lst[i] == x:
            return i
    return -1
```

Sous le **paradigme fonctionnel**, ceci peut se faire en OCaml via la fonction

```ocaml
let first lst x =
    let rec aux lst i =
        match lst with
            |[] -> -1
            |x' :: lst' when x = x' -> i
            |_ :: lst' -> aux lst' (i + 1)
    in
    aux lst 0
```

Ces deux fonctions renvoient plus précisément la **première occurrence** de `x` dans `lst`.

La valeur de retour `-1` est utilisée pour spécifier l'absence de `x` dans `lst`.

Sous le **paradigme logique**, en Prolog, nous pouvons considérer le prédicat

```
first([X | _], X, 0).

first([_ | Tail], X, Pos) :-
    first(Tail, X, Pos_1),
    Pos is Pos_1 + 1.
```

Ce prédicat `first(Lst, X, I)` **teste** si la valeur en l'indice `I` de la liste `Lst` est `X`.

La **résolution** de la **requête** `first([2, 10, 3, 10, 1, 1, 4, 10], 10, 1)` donne `true.`.

Nous pouvons observer que la résolution de `first([2, 10, 3, 10, 1, 1, 4, 10], 10, 3)` donne aussi `true`.

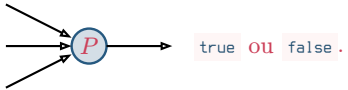
Dans ce paradigme, il est possible de laisser le troisième argument en tant que **variable**. Dans ce cas, lors de la résolution, **toutes les solutions** vont être engendrées.

Ainsi, la requête `first([2, 10, 3, 10, 1, 1, 4, 10], 10, I)` donne `I = 1 ; I = 3 ; I = 7 ; false.`

Ceci constitue un avantage énorme de ce paradigme.

1.2.5. Principes généraux de la programmation logique

La **programmation logique** est un paradigme de programmation dans lequel les objets de base sont les **clauses** et les **prédicats**. Un prédicat P est une entité qui accepte des valeurs en entrée et qui répond par `true` ou `false` :



Les principales **caractéristiques du paradigme fonctionnel** (absence d'effet secondaire, absence de boucle, utilisation de la récursivité) s'appliquent aussi au paradigme logique.

Il n'y a plus de notion de **sortie** aussi claire que dans les autres paradigmes : chaque entrée d'un prédicat peut être vue comme une sortie et réciproquement.

1.3. Initiation rapide à l'OCaml

1.3.1. Programmes

Un **programme OCaml** est un fichier texte d'extension `.ml`.

Il est structuré de la manière suivante :

1. suite de définitions (de types et de fonctions) ;
2. fonction principale.

La **fonction principale** est le **point d'entrée** du programme : c'est par elle que commence son exécution.

Celle-ci se définit via la syntaxe

```
let () =
    ...
```

Les `...` dénotent une expression.

Si `Program.ml` est un programme OCaml structuré selon la description précédente, il s'**exécute** via la commande

```
ocaml Program.ml
```

Il est également possible d'obtenir un **exécutable** via la commande

```
ocamlc Program.ml
```

Par défaut, l'exécutable construit se nomme `a.out`.

D'autres façons (plus évoluées) de programmer seront vues plus tard, incluant

- l'utilisation de l'interpréteur OCaml (potentiellement avec `utop`) ;
- l'utilisation du compilateur natif (`ocamlopt`) ;
- la programmation sur plusieurs fichiers (potentiellement avec `dune`).

1.3.2. Principes de base

Nous garderons en tête, dans notre initiation à l'OCaml, les principes suivants.

1. Un **programme** est une **expression**.
2. **Exécuter** un programme consiste à l'**évaluer**.
3. Le **résultat** de l'exécution d'un programme est sa **valeur**. Celle-ci est l'expression finale obtenue à l'issue du processus d'évaluation du programme.
4. Chaque **expression** possède un **unique type**.
5. Une expression doit respecter à la fois des **contraintes syntaxiques** et des **contraintes de type**.
6. Il n'y a **pas d'effet secondaire** en programmation fonctionnelle. Seule la valeur compte.
7. Cependant, nous allons assouplir la règle précédente pour nous autoriser les effets secondaires visant à réaliser des **entrées** et des **sorties**.

Exemples

```
let () =
    11 * 10 + 3
```

Ce programme a `113` comme valeur.

```
let () =
    let x = 11 * 10 + 3 in
    print_int x;
    print_newline ()
```

Ce programme a `()` comme valeur (c'est la valeur renvoyée par `print_newline ()`).

Il a comme effet secondaire d'afficher la chaîne de caractères `113` suivie d'un retour à la ligne.

```
let () =
    let x = read_int () in
    let y = 2 * x in
    print_string "double : ";
    print_int y;
    print_newline ()
```

Ce programme a `()` comme valeur.

Il a comme effet secondaire de lire un entier sur l'entrée standard et d'afficher la chaîne de caractères dénotant le double cet entier, suivie d'un retour à la ligne.

1.3.3. Structures principales

- **Liaison globale** d'un nom à une valeur :

```
let NAME = EXP
```

Ceci fait en sorte que dans toute la suite du programme, `NAME` soit un **alias** de l'expression `EXP`.

- **Liaison locale** d'un nom à une valeur :

```
let NAME = EXP_1 in EXP_2
```

Ceci fait en sorte que dans l'expression `EXP_2`, `NAME` soit un **alias** de l'expression `EXP_1`.

Exemples

```
let size = 32

let () =
    print_int size
```

Dans ce programme, une **liaison globale** de l'alias `size` à la valeur `32` est effectuée.

```
let () =
    let x = 14 + 2 in
    let y = x * 2 in
    print_int y
```

Dans ce programme, une **liaison locale** de l'alias `x` à la valeur `16` est effectuée dans une expression dans laquelle une **liaison locale** de l'alias `y` à la valeur `x * 2` est effectuée.

- **Définition d'une fonction** à plusieurs **paramètres** :

```
let NAME X_1 ... X_N = EXP
```

Ceci fait en sorte que dans toute la suite du programme, `NAME` soit une **fonction** qui peut être appelée avec `N` **arguments** (nous verrons plus loin qu'il sera dans certains cas possible et même tres utile d'appeler cette fonction avec un nombre différent d'arguments).

- **Définition récursive d'une fonction** à plusieurs paramètres :

```
let rec NAME X_1 ... X_N = EXP
```

Ceci fait en sorte que dans toute la suite du programme, `NAME` soit une **fonction** qui peut être appelée avec `N` arguments. Il est de plus possible d'appeler `NAME` dans `EXP`. Ceci peut donc potentiellement la rendre **récursive**.

Une fonction `f` s'**applique** à des arguments `a_1`, …, `a_n` en les y **juxtaposant** par `f a_1 ... a_n`.

Exemples

```
let sum_squares x y =
    x * x + y * y

let () =
    let x = sum_squares 8 (sum_squares 2 4) in
    print_int x
```

Dans ce programme une **définition** de fonction nommée `sum_squares` est effectuée.

Elle est ensuite utilisée deux fois dans la fonction principale.

```
let rec triangle n =
    if n = 0 then
        0
    else
        n + triangle (n - 1)

let () =
    let x = triangle 10 in
    print_int x
```

Dans ce programme, une **définition de fonction récursive** nommée `triangle` est effectuée.

Elle est ensuite utilisée une fois dans la fonction principale.

- **Conditionnelle** :

```
if EXP_1 then EXP_2 else EXP_3
```

Cette expression s'évalue en `EXP_2` si la condition `EXP_1` est vraie et en `EXP_3` sinon.

- **Filtrage de motif** :

```
match EXP with
    |PAT_1 -> EXP_1
    ...
    |PAT_N -> EXP_N
```

Cette expression s'évalue en `EXP_K` si `K` est le plus petit indice tel que l'expression `EXP` est filtrée par le motif `PAT_K`.

Exemples

```
if 244 mod 3 = 1 then 21 else 31
```

Cette expression s'évalue en `21` car la condition `244 mod 3 = 1` est vraie.

```
let one_or_two x =
    match x with
        |1 -> 1
        |2 -> 2
        |3 -> 2
        |4 -> 2
```

Cette fonction implante l'application $f : \{1, 2, 3, 4\} \to \{1, 2\}$ telle que $f(1) = 1$, $f(2) = 2$, $f(3) = 2$ et $f(4) = 2$.

Par exemple, `one_or_two 3` est une expression qui s'évalue en `2`.

1.3.4. Types de base

Les types de base sont

- le type `int` des **entiers**, exprimés en décimal et munis des opérateurs classiques ;
- le type `string` des **chaînes de caractères**, exprimées entre `"` ;
- le type `bool` des **booléens**, `true` et `false` ;
- le type `'a * 'b` des **couples**, exprimés entre parenthèses et séparés par une virgule.

Exemples

Quelques expressions

- de type `int` : `0`, `0 + 0`, `if 22 mod 2 = 0 then 8 + 1 else 9 * 2` ;
- de type `string` : `""`, `"10AB"`, `"ab" ^ "bba"` ;
- de type `bool` : `true`, `let b = false in b && true` ;
- de type `'a * 'b` : `(1, 2 + 2)`, `("ab", 11)`, `(1, ("ca", "ba"))`.