Paradigmes de Programmation : impératif, fonctionnel, objet. Exemples et applications

I. Concept de paradigme

Définition 1 **Un paradigme de programmation** correspond à un ensemble de concepts, pratiques et outils pour représenter et résoudre un problème sous forme de programme informatique.

Remarque 2 **Les langages de programmation** peuvent **favoriser** par leur syntaxe, sémantique ou librairies un ou plusieurs paradigmes de programmation. Tout paradigme peut cependant être utilisé dans tout langage généraliste.

	C	OCaml	Python
Impératif	✅	✅	✅
Fonctionnel	🚧	✅	⚠️
Objet	🚧	✅	✅

- ✅ : Supporté par le langage
- ⚠️ : Supporté par le langage, attention à la portée des variables
- 🚧 : Encodable, mais pas directement supporté par le langage

Remarque 3 Cette liste de paradigmes n'est pas exhaustive, il s'agît simplement des plus prédominants. On peut notamment citer le paradigme logique, qui consiste à indiquer ce que l'on souhaite calculer plutôt que la manière dont ce calcul s'opère. Ce paradigme est utilisé en SQL pour exprimer des requêtes.

II. Programmation impérative

Définition 4 [MITCH 4.4, p. 77] **Le paradigme impératif** voit un programme comme des *instructions* modifiant un *état*. Dans le paradigme impératif, il y a donc une distinction forte entre *expressions* s'évaluant vers des valeurs (qui peuvent être affectées à des variables), et *instructions*. L'instruction de base est *l'affectation*, qui modifie la valeur d'une variable.

Exemple 5 Les processeurs exécutent des programmes composés d'une suite d'instructions binaires, faisant changer registres et mémoire. Le langage d'assemblage, permettant de représenter les instructions machine, peut être qualifié de *langage impératif*.

Remarque 6 Conceptuellement, on peut rapprocher la programmation impérative du modèle de calcul des **machines de Turing**, ou du modèle de **machine de Von-Neumann**.

Définition 7 [MITCH 8.1.2, p. 206] Les instructions peuvent être combinées en les séquençant (en les exécutant les unes après les autres) ou avec des **structures de contrôle** telles que les conditions (`if then else`) et les boucles (`for`, `while`).

Remarque 8 En OCaml, la notion d'*instruction* n'existe pas vraiment: tout est expression ou déclaration. Le OCaml n'est donc pas un language fondamentalement impératif. Cependant, il existe en OCaml des structures mutables (e.g. références, tableaux) et leur valeur peut être changée par des expressions de type `unit` , qui se comportent donc comme des instructions. Il est tout à fait possible de faire de la programmation impérative en OCaml.

Exemple 9 **Séquence de deux affectations en C, OCaml et Python**

En C	En OCaml	En Python

`int x;`
`x = 2;`
`x = 2 * x;`
	`let x: int ref = ref 0 in`
	`x := 2;`
	`x := 2 * !x`
		
		`x = 2`
		`x = 2 * x`

III. Programmation fonctionnelle

A. Définition et contexte historique [MITCH 4.4.2]

Définition 10 **Le paradigme fonctionnel** voit un programme comme une *expression*, faisant intervenir une composition de *fonctions*.

Remarque 11 Conceptuellement, on peut rapprocher le paradigme fonctionnel du modèle de calcul introduit par Alonzo Church dans les années 1930, appelé **lambda-calcul**.

Contexte historique 12 [MITCH 3] **Lisp,** un des premiers langages fonctionnels, a été inventé au MIT dans les années 1950, pour des recherches sur l'intelligence artificielle et l'exécution symbolique.

B. Fonctionnel pur

Définition 13 On dit qu'une expression a des **effets de bord** si elle modifie l'état global du programme (e.g. changer la valeur d'une variable) ou de la machine (e.g. écriture dans un fichier).

Définition 14 [MITCH 4.4.2] **Une fonction pure** est une fonction n'ayant pas d'effets de bord. Un langage fonctionnel est dit **pur** s'il ne permet de définir que des fonctions pures.

Remarque 15 OCaml n'est pas un language fonctionnel pur.

Définition 16 [MITCH 4.4.2, p. 78] **La transparence référentielle** est le fait, pour une expression, de pouvoir être remplacée par la valeur vers laquelle elle s'évalue, sans changer le comportement du programme. Une fonction référentiellement transparente doit s'évaluer à la même valeur sur une même entrée, comme une fonction mathématique.

En particulier, si une fonction n'a pas d'effet de bord, alors elle sera référentiellement transparente.

Remarque 17 Cette propriété est utile lorsque l'on **raisonne** sur un programme, en particulier pour en **prouver la correction**. Si un programme est une composition fonctions pures, on pourra raisonner sur chaque fonction indépendamment. On parle de **compositionnalité**.

Remarque 18 Pour interagir avec l'utilisateur, un programme doit nécessairement causer des effets de bord. DEV

Définition 19 La programmation monadique peut permettre, dans les langages fonctionnels purs, d'encapsuler les effets de bords et garder la transparence référentielle.

C. Fonctions comme valeurs de première classe

Définition 20 [MITCH 7.4.1] Dans un langage, on dit que les **fonctions** sont des **valeurs de première classe** quand les fonctions:

- peuvent être définies dans n'importe quel contexte
- peuvent être passées en argument à d'autres fonctions
- peuvent être le résultat d'un appel de fonction

On parle de **fonction d'ordre supérieur** pour les fonctions prenant des fonctions en argument, et/ou renvoyant une fonction.

Exemple 21 La fonction `map` prend une fonction et une liste en arguments, et renvoie une liste. Les éléments de la liste en sortie correspondent à l'application de la fonction donnée aux éléments de la liste. Sa signature est

```
val map : ('a -> 'b) -> ('a list) -> 'b list
```

Remarque 22 [MITCH 4.2.3, p. 63] **Curryfication.** Chaque fonction a n arguments peut être vue comme ne prenant qu'un seul argument en entrée et renvoyant ou bien :

- une valeur si $n = 1$
- une fonction prenant $n - 1$ arguments en entrée sinon

Exemple 23 Appliquer la fonction `string_of_int` à la fonction `map` renvoie une nouvelle fonction. Appliquer cette nouvelle fonction à un argument donne le même résultat que appliquer `map` à `string_of_int` et à l'argument en question :

```
let map_to_string = map string_of_int
assert (map_to_string [0, 1, 2] = map string_of_int [0, 1, 2])
```

D. Récursivité terminale et optimisation [MITCH 7.3.4]

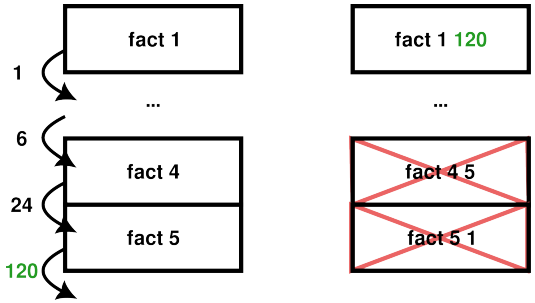
Définition 24 Si une fonction `f` appelle une fonction `g`, l'appel à `g` dans `f` est dit en **position terminale** si `f` renvoie la valeur de retour de `g` sans calculs supplémentaires.

Définition 25 Une fonction est récursive terminale si chaque appel récursif qu'elle fait est en position terminale.

Exemple 26 Factorielle récursive terminale

```
let fact (n: int) =
  let rec fact (n: int) (resultat: int) =
    if n <= 1 then resultat else fact (n - 1) (resultat * n)
  in fact n 1
```

Remarque 27 Une fonction récursive terminale peut être contenue dans un seul cadre de pile. La pile n'a alors pas besoin de grandir à chaque appel récursif, permettant d'éviter un débordement. Une telle fonction encode en réalité une **boucle**.



DEV

Remarque 28 [MITCH 8.3] La **programmation par passage de continuation** peut être utilisée par les compilateurs de langages fonctionnels pour transformer toute fonction récursive en récursive terminale.

IV. Programmation objet [MITCH 10]

Motivation 29 Faciliter la *collaboration* entre développeur·euses travaillant en parallèle, et rendre le code plus *réutilisable*.

Définition 30 **Le paradigme objet** voit un programme comme un ensemble de composants, entités ou concepts, appelés **objets**, interagissant entre eux.

Définition 31 **Les attributs** sont les données associées à un objet.

Définition 32 **Les méthodes** décrivent les façons dont on peut interagir avec un objet.

Définition 33 **L'encapsulation** est le principe selon lequel les définitions des attributs et méthodes d'un même objet devraient être au même endroit.

Remarque 34 Vu que beaucoup d'objets sont caractérisés par les mêmes attributs et méthodes, on ne définit pas directement des objets, mais on définit des **classes** (qui définissent des attributs et des méthodes). En *instantiant* les classes, on obtient des objets qui ont leur propre copie des attributs en question.

Définition 35 [MITCH p.278] **Les quatre principes de la programmation orientée objet** sont

- La **résolution dynamique** : le code exécuté par un appel de méthode dépend de l'objet sur lequel la méthode est appelée.
- L'**abstraction** : chaque objet (ou classe) expose une certaine **interface** spécifiant comment interagir avec lui. Les détails d'implémentation et certains attributs sont privés.
- Le **sous-typage** : si un objet `a` implémente toute l'interface d'un objet `b`, on doit pouvoir utiliser `a` là où `b` est attendu.
- L'**héritage** : une classe peut réutiliser la définition d'une autre classe, en l'étendant ou la modifiant si besoin.

Exemple 36 Code modélisant un restaurant:

```
class Plat: # Une classe modélisant les plats au menu
  def __init__(self):
    self.nb_commandes = 0
    # Un attribut : combien de foix le plat  a été commandé

  def prix(self) -> int : return 10 # Le prix par défaut d'un plat

  def commander(self) -> int : # Une méthode
    self.nb_commandes += 1
    return self.prix()

class Spaghetti(Plat): # Spaghetti hérite de Plat
  def prix(self): return 9 # Les spaghettis sont moins chers

  # Pas besoin de redéfinir la méthode commander

salade_cesar = Plat() # salade_cesar est une instance de Plat
spaghettis_bolognaise = Spaghetti()

class Client:
  def payer(p: Plat) -> None :
    self.porte_monaie -= p.commander() # Le prix dépend du plat
```

DEV

Remarque 37 Un langage de programmation avec héritage multiple doit choisir comment résoudre les appels de méthode, en particulier en cas d'« héritage en diamant ».

Paradigmes de Programmation : impératif, fonctionnel, objet. Exemples et applications

I. Concept de paradigme

1 Def [NAN] Un paradigme de programmation

2 Rem [NAN] Les langages de programmation

3 Rem [NAN] SQL au programme, mais pas dans cette leçon

II. Programmation impérative

4 Def [MITCH 4.4, p. 77] Le paradigme impératif

5 Ex [NAN] L'assembleur est impératif

6 Rem [NAN] Généalogie conceptuelle de l'impératif

7 Def Structures de contrôle

8 Rem [NAN] Le OCaml n'est pas impératif

9 Ex Séquence de deux affectations en C, OCaml et Python

III. Programmation fonctionnelle

A. Définition et contexte historique [MITCH 4.4.2]

10 Def Le paradigme fonctionnel

11 Rem [NAN] Généalogie conceptuelle du fonctionnel

12 Contexte historique [MITCH 3] Lisp,

B. Fonctionnel pur

13 Def Effets de bord

14 Def [MITCH 4.4.2] Une fonction pure

15 Rem [NAN] OCaml n'est pas un langage fonctionnel pur.

16 Def [MITCH 4.4.2, p. 78] La transparence référentielle

17 Rem

18 Rem Représentation des effets de bord en fonctionnel

19 Def La programmation monadique

C. Fonctions comme valeurs de première classe

20 Def Fonctions de première classe

21 Ex La fonction `map`

22 Rem [MITCH 4.2.3, p. 63]Curryfication.

23 Ex Curryfication

D. Récursivité terminale et optimisation [MITCH 7.3.4]

24 Def Appel en position terminal

25 Def Une fonction est récursive terminale

26 Ex Factorielle récursive terminale

27 Rem

28 Rem Continuations

IV. Programmation objet [MITCH 10]

29 Motiv Motivation objet: dev collab

30 Def Le paradigme objet

31 Def Les attributs

32 Def Les méthodes

33 Def L'encapsulation

34 Rem Classe

35 Def [MITCH p.278] Les quatre principes de la programmation orientée objet

36 Ex [NAN] Orienté objet

37 Rem Héritage multiple

Notes sur les Devs

- Programmation par continuation → Bien mais attention à erreurs des devs précédents. Regarder cours de Benoît en ligne: https://people.rennes.inria.fr/Benoit.Montagu/courses/agreg_prog_lang/cours5_handout.pdf). Historiquement, c'est une repr intermédiaire pour les compilos, c'est important pour bien justifier et pas juste « faire du code rigolo ».
- Autre dev possible (2024) Un même algo dans les 3 paradigmes
- Dev desing pattern (e.g. visiteur) → très bien si à l'aise dessus
- Inférence de types/unification → hors sujet

Notes générales

- Les notions de récursivité ne sont pas spécifiques au fonctionnel. Attention à la déf: notion plus sémantique que syntaxique
- La terminologie en prog objet est variable. Par exemple, un attribut de classe en Python dénote un attribut statique de la classe. Ça peut être à relever dans la présentation du plan.
- En objet, on donne les exemples en Python, mais on parle de l'objet en général. L'objet Python est particulier, le jury sait que ce n'est pas le paroxysme de l'orienté objet… mais au programme
- En général, **prendre les defs dans un livre** et éviter d'inventer les siennes sur le tas.
- Note sur le polymorphisme : il y en a souvent dans les plans des années précédentes. Pas vraiment relié à un paradigme en particulier, et sens différent par paradigme (polymorphisme objet ≠ polymorphisme fonctionnel par exemple, Python peut être vu comme « naturellement polymorphe » puisque typage dynamique, bref c'est un peu bourbier)

Bibliographie

[MITCH] J. C. Mitchell, *Concepts in programming languages.*

Aloïs Rautureau & Santiago Sara Bautista