Programmation Fonctionnelle Complète
Applications en Python et OCaml

Votre Nom

3 novembre 2025

Table des matières

1 Introduction aux Bases de la Programmation Fonctionnelle

1.1 Concepts Fondamentaux

— **Immutabilité** : Les données ne sont pas modifiables
— **Pureté** : Pas d'effets de bord
— **Fonctions de première classe** : Les fonctions sont des valeurs
— **Récursion** : Alternative aux boucles

1.2 Exemple en Python

```python
# Immutabilit  avec tuple
point = (1, 2)
nouveau_point = (point[0] + 1, point[1] + 1)

# Fonction pure
def ajouter(a, b):
    return a + b

# R cursion
def factorielle(n):
    return 1 if n == 0 else n * factorielle(n-1)

print(f"Point: {point} -> Nouveau: {nouveau_point}")
print(f"Addition: {ajouter(5, 3)}")
print(f"Factorielle de 5: {factorielle(5)}")
```

Listing 1 – Bases en Python

1.3 Exemple en OCaml

```
(* Immutabilit  par d faut *)
let point = (1, 2)
let nouveau_point = (fst point + 1, snd point + 1)

(* Fonction pure *)
let ajouter a b = a + b

(* R cursion *)
let rec factorielle n =
    if n = 0 then 1 else n * factorielle (n - 1)

let () =
    Printf.printf "Point: (%d, %d) -> Nouveau: (%d, %d)\n"
        (fst point) (snd point) (fst nouveau_point) (snd
           nouveau_point);
    Printf.printf "Addition: %d\n" (ajouter 5 3);
    Printf.printf "Factorielle de 5: %d\n" (factorielle 5)
```

Listing 2 – Bases en OCaml

2 Fonctions et Composition

2.1 Fonctions d'Ordre Supérieur

2.1.1 Exemple en Python

```
def appliquer_deux_fois(f, x):
    return f(f(x))

def carre(x):
    return x * x

def inc(x):
    return x + 1

# Utilisation
resultat1 = appliquer_deux_fois(carre, 3)  # 81
resultat2 = appliquer_deux_fois(inc, 5)    # 7

print(f"Carre deux fois de 3: {resultat1}")
print(f"Increment deux fois de 5: {resultat2}")

# Fonctions anonymes
resultat3 = appliquer_deux_fois(lambda x: x * 2, 2)  # 8
print(f"Double deux fois de 2: {resultat3}")
```

Listing 3 – Fonctions d'ordre supérieur en Python

2.1.2 Exemple en OCaml

```
let appliquer_deux_fois f x = f (f x)

let carre x = x * x
let inc x = x + 1

(* Utilisation *)
let resultat1 = appliquer_deux_fois carre 3  (* 81 *)
let resultat2 = appliquer_deux_fois inc 5    (* 7 *)

let () =
    Printf.printf "Carre deux fois de 3: %d\n" resultat1;
    Printf.printf "Increment deux fois de 5: %d\n" resultat2

(* Fonctions anonymes *)
let resultat3 = appliquer_deux_fois (fun x -> x * 2) 2  (* 8 *)
let () = Printf.printf "Double deux fois de 2: %d\n" resultat3
```

Listing 4 – Fonctions d'ordre supérieur en OCaml

2.2 Composition de Fonctions

2.2.1 Exemple en Python

```
from functools import reduce

def composer(*fonctions):
    """Compose plusieurs fonctions de droite   gauche"""
    return reduce(lambda f, g: lambda x: f(g(x)), fonctions)

# Fonctions de base
def ajouter_1(x): return x + 1
def multiplier_2(x): return x * 2
def carre(x): return x ** 2

# Composition
transformation = composer(carre, multiplier_2, ajouter_1)
resultat = transformation(3)  # ((3+1)*2)^2 = 64

print(f"Transformation de 3: {resultat}")

# Alternative avec pipe
def pipe(*fonctions):
    """Pipe plusieurs fonctions de gauche   droite"""
    return reduce(lambda f, g: lambda x: g(f(x)), fonctions)

transformation_pipe = pipe(ajouter_1, multiplier_2, carre)
resultat_pipe = transformation_pipe(3)  # ((3+1)*2)^2 = 64
print(f"Transformation pipe de 3: {resultat_pipe}")
```

Listing 5 – Composition en Python

2.2.2 Exemple en OCaml

```
(* Op rateur de composition standard *)
let ( >> ) f g x = g (f x)
let ( << ) f g x = f (g x)

(* Fonctions de base *)
let ajouter_1 x = x + 1
let multiplier_2 x = x * 2
let carre x = x * x

(* Composition droite    gauche *)
let transformation = ajouter_1 >> multiplier_2 >> carre
let resultat = transformation 3  (* ((3+1)*2)^2 = 64 *)

(* Composition gauche    droite *)
let transformation2 = carre << multiplier_2 << ajouter_1
let resultat2 = transformation2 3  (* ((3+1)*2)^2 = 64 *)

let () =
    Printf.printf "Transformation de 3: %d\n" resultat;
    Printf.printf "Transformation2 de 3: %d\n" resultat2
```

Listing 6 – Composition en OCaml

3 Currying et Application Partielle

3.1 Exemple en Python

```
from functools import partial

# Currying manuel
def addition_curry(a):
    def inner(b):
        return a + b
    return inner

# Utilisation
ajouter_5 = addition_curry(5)
resultat = ajouter_5(3)  # 8

print(f"Addition curry: {resultat}")

# Avec functools.partial
def multiplier(a, b):
    return a * b

multiplier_par_3 = partial(multiplier, 3)
resultat2 = multiplier_par_3(4)  # 12

```

```python
print(f"Multiplication partielle: {resultat2}")

# Currying automatique
def curry(f):
    """Transforme une fonction en fonction curryfi e"""
    def curried(*args):
        if len(args) >= f.__code__.co_argcount:
            return f(*args)
        else:
            return lambda *more_args: curried(*(args + more_args))
    return curried

@curry
def addition_trois_nombres(a, b, c):
    return a + b + c

add_5 = addition_trois_nombres(5)
add_5_3 = add_5(3)
resultat3 = add_5_3(2)  # 10

print(f"Addition trois nombres curry: {resultat3}")
```

Listing 7 – Currying en Python

3.2 Exemple en OCaml

```ocaml
(* Currying natif en OCaml *)
let addition a b = a + b

(* Application partielle automatique *)
let ajouter_5 = addition 5
let resultat = ajouter_5 3  (* 8 *)

let () = Printf.printf "Addition curry: %d\n" resultat

(* Fonction    trois arguments *)
let addition_trois_nombres a b c = a + b + c

(* Currying multiple *)
let add_5 = addition_trois_nombres 5
let add_5_3 = add_5 3
let resultat2 = add_5_3 2  (* 10 *)

let () = Printf.printf "Addition trois nombres curry: %d\n"
   resultat2

(* Utilisation pratique *)
let nombres = [1; 2; 3; 4; 5]
let ajouter_3_liste = List.map (addition 3)
let resultat_liste = ajouter_3_liste nombres  (* [4;5;6;7;8] *)

```

```
let () =
    Printf.printf "Liste originale: [";
    List.iter (Printf.printf "%d; ") nombres;
    Printf.printf "]\n";
    Printf.printf "Liste apres ajout de 3: [";
    List.iter (Printf.printf "%d; ") resultat_liste;
    Printf.printf "]\n"
```

Listing 8 – Currying en OCaml

4 Récursion

4.1 Récursion Simple vs Récursion Terminale

4.1.1 Exemple en Python

```
# R cursion simple (peut causer stack overflow)
def somme_liste_simple(lst):
    if not lst:
        return 0
    return lst[0] + somme_liste_simple(lst[1:])

# R cursion terminale (optimis e)
def somme_liste_terminale(lst, acc=0):
    if not lst:
        return acc
    return somme_liste_terminale(lst[1:], acc + lst[0])

# R cursion avec pattern matching style
def somme_liste_pattern(lst):
    match lst:
        case []: return 0
        case [x, *reste]: return x + somme_liste_pattern(reste)

# Utilisation
nombres = [1, 2, 3, 4, 5]
print(f"Somme simple: {somme_liste_simple(nombres)}")
print(f"Somme terminale: {somme_liste_terminale(nombres)}")
print(f"Somme pattern: {somme_liste_pattern(nombres)}")

# R cursion pour Fibonacci
def fibonacci(n):
    def fib_helper(n, a, b):
        if n == 0:
            return a
        elif n == 1:
            return b
        else:
            return fib_helper(n - 1, b, a + b)
    return fib_helper(n, 0, 1)

```

```python
print(f"Fibonacci de 10: {fibonacci(10)}")
```

Listing 9 – Récursion en Python

4.1.2 Exemple en OCaml

```
(* Recursion simple *)
let rec somme_liste_simple = function
    | [] -> 0
    | tete :: queue -> tete + somme_liste_simple queue

(* Recursion terminale optimisee *)
let somme_liste_terminale lst =
    let rec aux acc = function
        | [] -> acc
        | tete :: queue -> aux (acc + tete) queue
    in aux 0 lst

(* Pattern matching complet *)
let rec somme_liste_pattern lst =
    match lst with
    | [] -> 0
    | [x] -> x
    | x :: y :: reste -> x + y + somme_liste_pattern reste

(* Utilisation *)
let nombres = [1; 2; 3; 4; 5]
let () =
    Printf.printf "Somme simple: %d\n" (somme_liste_simple nombres)
       ;
    Printf.printf "Somme terminale: %d\n" (somme_liste_terminale
       nombres);
    Printf.printf "Somme pattern: %d\n" (somme_liste_pattern
       nombres)

(* Fibonacci avec recursion terminale *)
let fibonacci n =
    let rec fib_helper n a b =
        if n = 0 then a
        else if n = 1 then b
        else fib_helper (n - 1) b (a + b)
    in fib_helper n 0 1

let () = Printf.printf "Fibonacci de 10: %d\n" (fibonacci 10)
```

Listing 10 – Récursion en OCaml

5 Types Option/Maybe et Gestion des Valeurs Null

5.1 Exemple en Python

```python
from typing import TypeVar, Generic, Optional
from abc import ABC, abstractmethod

T = TypeVar('T')
U = TypeVar('U')

class Maybe(Generic[T], ABC):
    @abstractmethod
    def map(self, f) -> 'Maybe[U]': pass

    @abstractmethod
    def flat_map(self, f) -> 'Maybe[U]': pass

    @abstractmethod
    def get_or_else(self, default: T) -> T: pass

class Some(Maybe[T]):
    def __init__(self, value: T):
        self.value = value

    def map(self, f):
        return Some(f(self.value))

    def flat_map(self, f):
        return f(self.value)

    def get_or_else(self, default: T):
        return self.value

    def __str__(self):
        return f"Some({self.value})"

class Nothing(Maybe[T]):
    def map(self, f):
        return Nothing()

    def flat_map(self, f):
        return Nothing()

    def get_or_else(self, default: T):
        return default

    def __str__(self):
        return "Nothing"

# Utilisation pratique
def trouver_element(lst, predicate):
```

```python
    for item in lst:
        if predicate(item):
            return Some(item)
    return Nothing()

def diviser_securise(a: float, b: float) -> Maybe[float]:
    if b == 0:
        return Nothing()
    return Some(a / b)

# Exemple d'utilisation
nombres = [1, 2, 3, 4, 5]

# Cha nage d'op rations
resultat = (trouver_element(nombres, lambda x: x > 3)
            .map(lambda x: x * 2)
            .flat_map(lambda x: diviser_securise(x, 2))
            .get_or_else(0))

print(f"R sultat : {resultat}")  # 4.0

# Avec les Optional de Python
def diviser_optional(a: float, b: float) -> Optional[float]:
    return a / b if b != 0 else None

def traiter_optional(x: Optional[float]) -> Optional[float]:
    return None if x is None else x * 2

resultat_optional = traiter_optional(diviser_optional(8, 2))
print(f"R sultat optional : {resultat_optional}")
```

Listing 11 – Option/Maybe en Python

5.2 Exemple en OCaml

```ocaml
(* Le type option est natif en OCaml *)

(* Fonctions utilitaires pour option *)
let map_option f = function
    | Some x -> Some (f x)
    | None -> None

let flat_map_option f = function
    | Some x -> f x
    | None -> None

let get_or_else default = function
    | Some x -> x
    | None -> default

(* Utilisation pratique *)
```

```ocaml
let trouver_element predicate lst =
    let rec aux = function
        | [] -> None
        | tete :: queue ->
            if predicate tete then Some tete
            else aux queue
    in aux lst

let diviser_securise a b =
    if b = 0 then None else Some (a / b)

(* Exemple d'utilisation *)
let nombres = [1; 2; 3; 4; 5]

(* Cha nage d'op rations avec l'operateur |> *)
let resultat =
    trouver_element (fun x -> x > 3) nombres
    |> map_option (fun x -> x * 2)
    |> flat_map_option (fun x -> diviser_securise x 2)
    |> get_or_else 0

let () = Printf.printf "R sultat : %d\n" resultat  (* 4 *)

(* Pattern matching sur les options *)
let afficher_option = function
    | Some x -> Printf.printf "Valeur : %d\n" x
    | None -> print_endline "Aucune valeur"

let () = afficher_option (Some 42)
let () = afficher_option None

(* Composition d'op rations avec options *)
let calcul_complexe x y =
    diviser_securise x y
    |> map_option (fun z -> z + 1.0)
    |> map_option (fun z -> z *. 2.0)

let () =
    match calcul_complexe 10.0 2.0 with
    | Some result -> Printf.printf "R sultat complexe : %f\n"
       result
    | None -> print_endline "Calcul impossible"
```

Listing 12 – Option en OCaml

6 Type Either pour la Gestion d'Erreurs

6.1 Exemple en Python

```python
from typing import Generic, TypeVar, Union
```

```python
from abc import ABC, abstractmethod

L = TypeVar('L')
R = TypeVar('R')

class Either(Generic[L, R], ABC):
    @abstractmethod
    def map(self, f) -> 'Either[L, R]': pass

    @abstractmethod
    def flat_map(self, f) -> 'Either[L, R]': pass

    @abstractmethod
    def is_left(self) -> bool: pass

    @abstractmethod
    def is_right(self) -> bool: pass

class Left(Either[L, R]):
    def __init__(self, value: L):
        self.value = value

    def map(self, f):
        return Left(self.value)

    def flat_map(self, f):
        return Left(self.value)

    def is_left(self):
        return True

    def is_right(self):
        return False

    def __str__(self):
        return f"Left({self.value})"

class Right(Either[L, R]):
    def __init__(self, value: R):
        self.value = value

    def map(self, f):
        return Right(f(self.value))

    def flat_map(self, f):
        return f(self.value)

    def is_left(self):
        return False

    def is_right(self):
```

```python
        return True

    def __str__(self):
        return f"Right({self.value})"

# Utilisation pour la gestion d'erreurs
def parse_int(s: str) -> Either[str, int]:
    try:
        return Right(int(s))
    except ValueError:
        return Left(f"Impossible de parser '{s}' en entier")

def diviser_either(a: int, b: int) -> Either[str, float]:
    if b == 0:
        return Left("Division par zero")
    return Right(a / b)

# Cha nage d'op rations
def calcul_complexe(s1: str, s2: str) -> Either[str, float]:
    return (parse_int(s1)
            .flat_map(lambda x:
                parse_int(s2)
                .flat_map(lambda y:
                    diviser_either(x, y)
                    .map(lambda z: z * 2))))

# Exemples
resultats = [
    calcul_complexe("10", "2"),    # Right(10.0)
    calcul_complexe("10", "0"),    # Left("Division par zero")
    calcul_complexe("abc", "2")    # Left("Impossible de parser 'abc
       ' en entier")
]

for resultat in resultats:
    print(f"Resultat: {resultat}")

# Pattern matching style
def traiter_either(e: Either[str, float]):
    if e.is_right():
        print(f"Succes: {e.value}")
    else:
        print(f"Erreur: {e.value}")

for resultat in resultats:
    traiter_either(resultat)
```

Listing 13 – Either en Python

6.2 Exemple en OCaml

```
(* D finition du type Either *)
type ('l, 'r) either =
    | Left of 'l
    | Right of 'r

(* Fonctions utilitaires *)
let map_either f = function
    | Right x -> Right (f x)
    | Left e -> Left e

let flat_map_either f = function
    | Right x -> f x
    | Left e -> Left e

let is_left = function
    | Left _ -> true
    | Right _ -> false

let is_right = function
    | Left _ -> false
    | Right _ -> true

(* Utilisation pour la gestion d'erreurs *)
let parse_int s =
    try Right (int_of_string s)
    with Failure _ -> Left ("Impossible de parser '" ^ s ^ "' en
       entier")

let diviser_either a b =
    if b = 0 then Left "Division par zero"
    else Right (float_of_int a /. float_of_int b)

(* Cha nage d'op rations *)
let calcul_complexe s1 s2 =
    parse_int s1
    |> flat_map_either (fun x ->
        parse_int s2
        |> flat_map_either (fun y ->
            diviser_either x y
            |> map_either (fun z -> z *. 2.0)))

(* Exemples *)
let resultats = [
    calcul_complexe "10" "2";    (* Right 10.0 *)
    calcul_complexe "10" "0";    (* Left "Division par zero" *)
    calcul_complexe "abc" "2"    (* Left "Impossible de parser 'abc'
        en entier" *)
]

let afficher_resultat = function
    | Right x -> Printf.printf "Succes: %f\n" x
```

```
    | Left e -> Printf.printf "Erreur : %s\n" e

let () = List.iter afficher_resultat resultats

(* Combinateurs pour Either *)
let lift2_either f either1 either2 =
    match either1, either2 with
    | Right x, Right y -> Right (f x y)
    | Left e, _ -> Left e
    | _, Left e -> Left e

(* Exemple avec lift2 *)
let addition_either = lift2_either (+)
let multiplication_either = lift2_either ( * )

let exemple_lift2 =
    let r1 = parse_int "5" in
    let r2 = parse_int "3" in
    addition_either r1 r2  (* Right 8 *)

let () =
    match exemple_lift2 with
    | Right x -> Printf.printf "Addition avec lift2 : %d\n" x
    | Left e -> Printf.printf "Erreur : %s\n" e
```

Listing 14 – Either en OCaml

7 Listes et Listes Chaînées

7.1 Exemple en Python

```
from typing import TypeVar, Generic, Optional
from abc import ABC, abstractmethod

T = TypeVar('T')

class Liste(Generic[T], ABC):
    @abstractmethod
    def est_vide(self) -> bool: pass

    @abstractmethod
    def tete(self) -> T: pass

    @abstractmethod
    def queue(self) -> 'Liste[T]': pass

    @abstractmethod
    def __len__(self) -> int: pass

class ListeVide(Liste[T]):
```

```
20      def est_vide(self): return True
21      def tete(self): raise ValueError("Liste vide")
22      def queue(self): raise ValueError("Liste vide")
23      def __len__(self): return 0
24      def __str__(self): return "[]"
25
26  class ListeCons(Liste[T]):
27      def __init__(self, tete: T, queue: Liste[T]):
28          self._tete = tete
29          self._queue = queue
30
31      def est_vide(self): return False
32      def tete(self): return self._tete
33      def queue(self): return self._queue
34      def __len__(self): return 1 + len(self._queue)
35      def __str__(self):
36          return f"{self._tete}::{str(self._queue)}"
37
38  # Fonctions utilitaires
39  def liste_vide(): return ListeVide()
40
41  def cons(tete: T, queue: Liste[T]) -> Liste[T]:
42      return ListeCons(tete, queue)
43
44  def from_list(python_list: list[T]) -> Liste[T]:
45      if not python_list:
46          return liste_vide()
47      return cons(python_list[0], from_list(python_list[1:]))
48
49  def to_list(liste: Liste[T]) -> list[T]:
50      if liste.est_vide():
51          return []
52      return [liste.tete()] + to_list(liste.queue())
53
54  # Op rations fonctionnelles sur les listes
55  def map_liste(f, liste: Liste[T]) -> Liste[T]:
56      if liste.est_vide():
57          return liste_vide()
58      return cons(f(liste.tete()), map_liste(f, liste.queue()))
59
60  def filter_liste(predicate, liste: Liste[T]) -> Liste[T]:
61      if liste.est_vide():
62          return liste_vide()
63      elif predicate(liste.tete()):
64          return cons(liste.tete(), filter_liste(predicate, liste.
                queue()))
65      else:
66          return filter_liste(predicate, liste.queue())
67
68  def reduce_liste(f, acc, liste: Liste[T]):
69      if liste.est_vide():
```

```
        return acc
    return reduce_liste(f, f(acc, liste.tete()), liste.queue())

# Utilisation
ma_liste = from_list([1, 2, 3, 4, 5])
print(f"Liste originale: {ma_liste}")

liste_doublee = map_liste(lambda x: x * 2, ma_liste)
print(f"Liste doublee: {liste_doublee}")

liste_filtree = filter_liste(lambda x: x % 2 == 0, ma_liste)
print(f"Liste filtree (pairs): {liste_filtree}")

somme = reduce_liste(lambda acc, x: acc + x, 0, ma_liste)
print(f"Somme: {somme}")

# Avec les listes Python natives (approche fonctionnelle)
nombres = [1, 2, 3, 4, 5]
double = list(map(lambda x: x * 2, nombres))
pairs = list(filter(lambda x: x % 2 == 0, nombres))
somme_native = sum(nombres)

print(f"Double native: {double}")
print(f"Pairs native: {pairs}")
print(f"Somme native: {somme_native}")
```

Listing 15 – Listes fonctionnelles en Python

7.2 Exemple en OCaml

```
(* Les listes sont natives en OCaml *)

(* Fonctions de base *)
let rec longueur = function
    | [] -> 0
    | _ :: queue -> 1 + longueur queue

let rec map f = function
    | [] -> []
    | tete :: queue -> f tete :: map f queue

let rec filter predicate = function
    | [] -> []
    | tete :: queue ->
        if predicate tete then tete :: filter predicate queue
        else filter predicate queue

let rec fold_left f acc = function
    | [] -> acc
    | tete :: queue -> fold_left f (f acc tete) queue

```

```
let rec fold_right f lst acc =
    match lst with
    | [] -> acc
    | tete :: queue -> f tete (fold_right f queue acc)

(* Utilisation *)
let nombres = [1; 2; 3; 4; 5]

let double = map (fun x -> x * 2) nombres
let pairs = filter (fun x -> x mod 2 = 0) nombres
let somme = fold_left ( + ) 0 nombres
let produit = fold_right ( * ) nombres 1

let () =
    Printf.printf "Liste originale: [";
    List.iter (Printf.printf "%d; ") nombres;
    Printf.printf "]\n";
    Printf.printf "Double: [";
    List.iter (Printf.printf "%d; ") double;
    Printf.printf "]\n";
    Printf.printf "Pairs: [";
    List.iter (Printf.printf "%d; ") pairs;
    Printf.printf "]\n";
    Printf.printf "Somme: %d\n" somme;
    Printf.printf "Produit: %d\n" produit

(* Pattern matching avance *)
let rec somme_pairs = function
    | [] -> 0
    | x :: _ when x mod 2 = 0 -> x  (* Premier element pair *)
    | _ :: queue -> somme_pairs queue

let rec trouver predicate = function
    | [] -> None
    | tete :: queue when predicate tete -> Some tete
    | _ :: queue -> trouver predicate queue

(* Listes en comprehension style *)
let carres_pairs =
    nombres
    |> List.filter (fun x -> x mod 2 = 0)
    |> List.map (fun x -> x * x)

let () =
    Printf.printf "Carres des pairs: [";
    List.iter (Printf.printf "%d; ") carres_pairs;
    Printf.printf "]\n"

(* Concat nation fonctionnelle *)
let rec concat lst1 lst2 =
    match lst1 with
```

```
    | [] -> lst2
    | tete :: queue -> tete :: concat queue lst2

let liste_combinee = concat nombres double
let () =
    Printf.printf "Liste combinee: [";
    List.iter (Printf.printf "%d; ") liste_combinee;
    Printf.printf "]\n"
```

Listing 16 – Listes en OCaml

8 Types Algébriques de Données (ADT) et Pattern Matching

8.1 Exemple en Python

```python
from typing import Union, Optional
from dataclasses import dataclass
from enum import Enum

# ADT pour les formes g om triques
class Couleur(Enum):
    ROUGE = "rouge"
    VERT = "vert"
    BLEU = "bleu"

@dataclass
class Point:
    x: float
    y: float

# Types alg briques avec Union
Forme = Union['Cercle', 'Rectangle', 'Groupe']

@dataclass
class Cercle:
    centre: Point
    rayon: float
    couleur: Couleur

@dataclass
class Rectangle:
    coin_sup_gauche: Point
    largeur: float
    hauteur: float
    couleur: Couleur

@dataclass
class Groupe:
    formes: list[Forme]
```

```python

# Pattern matching avec match/case (Python 3.10+)
def aire_forme(forme: Forme) -> float:
    match forme:
        case Cercle(centre=_, rayon=r, couleur=_):
            return 3.14159 * r * r
        case Rectangle(coin_sup_gauche=_, largeur=l, hauteur=h,
           couleur=_):
            return l * h
        case Groupe(formes=formes_liste):
            return sum(aire_forme(f) for f in formes_liste)
        case _:
            raise ValueError("Forme inconnue")

def deplacer_forme(forme: Forme, dx: float, dy: float) -> Forme:
    match forme:
        case Cercle(centre=Point(x, y), rayon=r, couleur=c):
            return Cercle(Point(x + dx, y + dy), r, c)
        case Rectangle(coin_sup_gauche=Point(x, y), largeur=l,
           hauteur=h, couleur=c):
            return Rectangle(Point(x + dx, y + dy), l, h, c)
        case Groupe(formes=formes_liste):
            return Groupe([deplacer_forme(f, dx, dy) for f in
               formes_liste])
        case _:
            raise ValueError("Forme inconnue")

# ADT pour les expressions arithm tiques
Expression = Union['Nombre', 'Addition', 'Multiplication']

@dataclass
class Nombre:
    valeur: int

@dataclass
class Addition:
    gauche: Expression
    droite: Expression

@dataclass
class Multiplication:
    gauche: Expression
    droite: Expression

def evaluer_expression(expr: Expression) -> int:
    match expr:
        case Nombre(valeur=v):
            return v
        case Addition(gauche=g, droite=d):
            return evaluer_expression(g) + evaluer_expression(d)
        case Multiplication(gauche=g, droite=d):
```

```
            return evaluer_expression(g) * evaluer_expression(d)
        case _:
            raise ValueError("Expression inconnue")

# Utilisation
cercle = Cercle(Point(0, 0), 5, Couleur.ROUGE)
rectangle = Rectangle(Point(1, 1), 4, 3, Couleur.VERT)
groupe = Groupe([cercle, rectangle])

print(f"Aire du cercle: {aire_forme(cercle):.2f}")
print(f"Aire du rectangle: {aire_forme(rectangle)}")
print(f"Aire du groupe: {aire_forme(groupe):.2f}")

# Expression arithm tique: (2 + 3) * 4
expression = Multiplication(
    Addition(Nombre(2), Nombre(3)),
    Nombre(4)
)

print(f"  valuation  de (2+3)*4: {evaluer_expression(expression)}")
```

Listing 17 – ADT et Pattern Matching en Python

8.2 Exemple en OCaml

```
(* ADT pour les formes g om triques *)
type couleur = Rouge | Vert | Bleu

type point = { x : float; y : float }

type forme =
    | Cercle of { centre : point; rayon : float; couleur : couleur }
    | Rectangle of { coin_sup_gauche : point; largeur : float; hauteur : float; couleur : couleur }
    | Groupe of forme list

(* Pattern matching pour les formes *)
let aire_forme forme =
    match forme with
    | Cercle { centre = _; rayon = r; couleur = _ } ->
        3.14159 *. r *. r
    | Rectangle { coin_sup_gauche = _; largeur = l; hauteur = h; couleur = _ } ->
        l *. h
    | Groupe formes ->
        List.fold_left (fun acc f -> acc +. aire_forme f) 0.0 formes

let deplacer_forme forme dx dy =
    match forme with
```

```
23      | Cercle { centre = { x; y }; rayon = r; couleur = c } ->
24          Cercle { centre = { x = x +. dx; y = y +. dy }; rayon = r;
                couleur = c }
25      | Rectangle { coin_sup_gauche = { x; y }; largeur = l; hauteur
            = h; couleur = c } ->
26          Rectangle { coin_sup_gauche = { x = x +. dx; y = y +. dy };
                largeur = l; hauteur = h; couleur = c }
27      | Groupe formes ->
28          Groupe (List.map (fun f -> deplacer_forme f dx dy) formes)
29
30  (* ADT pour les expressions arithm tiques *)
31  type expression =
32      | Nombre of int
33      | Addition of expression * expression
34      | Multiplication of expression * expression
35
36  let rec evaluer_expression expr =
37      match expr with
38      | Nombre n -> n
39      | Addition (g, d) -> evaluer_expression g + evaluer_expression
            d
40      | Multiplication (g, d) -> evaluer_expression g *
            evaluer_expression d
41
42  (* Utilisation *)
43  let cercle = Cercle { centre = { x = 0.0; y = 0.0 }; rayon = 5.0;
        couleur = Rouge }
44  let rectangle = Rectangle { coin_sup_gauche = { x = 1.0; y = 1.0 };
        largeur = 4.0; hauteur = 3.0; couleur = Vert }
45  let groupe = Groupe [cercle; rectangle]
46
47  let () =
48      Printf.printf "Aire␣du␣cercle:␣%.2f\n" (aire_forme cercle);
49      Printf.printf "Aire␣du␣rectangle:␣%.2f\n" (aire_forme rectangle
            );
50      Printf.printf "Aire␣du␣groupe:␣%.2f\n" (aire_forme groupe)
51
52  (* Expression arithm tique: (2 + 3) * 4 *)
53  let expression = Multiplication (Addition (Nombre 2, Nombre 3),
        Nombre 4)
54
55  let () = Printf.printf "  valuation  ␣de␣(2+3)*4:␣%d\n" (
        evaluer_expression expression)
56
57  (* Pattern matching avance avec gardes *)
58  let classifier_forme forme =
59      match forme with
60      | Cercle { rayon = r; _ } when r > 10.0 -> "Grand␣cercle"
61      | Cercle { rayon = r; _ } when r < 1.0 -> "Petit␣cercle"
62      | Cercle _ -> "Cercle␣moyen"
```

```
    | Rectangle { largeur = l; hauteur = h; _ } when l = h -> "Carre"
    | Rectangle _ -> "Rectangle"
    | Groupe formes -> Printf.sprintf "Groupe de %d formes" (List.length formes)

let () =
    Printf.printf "Classification du cercle: %s\n" (classifier_forme cercle);
    Printf.printf "Classification du rectangle: %s\n" (classifier_forme rectangle)
```

Listing 18 – ADT et Pattern Matching en OCaml

9 Théorie des Catégories : Magma, Semigroup, Monoid

9.1 Exemple en Python

```python
from typing import TypeVar, Generic, Callable
from abc import ABC, abstractmethod

T = TypeVar('T')

# Magma: ensemble avec une operation binaire
class Magma(Generic[T], ABC):
    @abstractmethod
    def combine(self, other: T) -> T: pass

# Semigroup: Magma avec associativite
class Semigroup(Magma[T], ABC):
    pass  # L'associativite est une propriete, pas une methode

# Monoid: Semigroup avec element neutre
class Monoid(Semigroup[T], ABC):
    @abstractmethod
    @classmethod
    def empty(cls) -> T: pass

# Implementations concretes
class Somme(Monoid[int]):
    def __init__(self, valeur: int):
        self.valeur = valeur

    def combine(self, other: 'Somme') -> 'Somme':
        return Somme(self.valeur + other.valeur)

    @classmethod
    def empty(cls) -> 'Somme':
        return Somme(0)

```

```
33      def __str__(self):
34          return f"Somme({self.valeur})"
35
36  class Produit(Monoid[int]):
37      def __init__(self, valeur: int):
38          self.valeur = valeur
39
40      def combine(self, other: 'Produit') -> 'Produit':
41          return Produit(self.valeur * other.valeur)
42
43      @classmethod
44      def empty(cls) -> 'Produit':
45          return Produit(1)
46
47      def __str__(self):
48          return f"Produit({self.valeur})"
49
50  # Monoid pour les listes
51  class ListeMonoid(Monoid[list[T]]):
52      def __init__(self, elements: list[T]):
53          self.elements = elements
54
55      def combine(self, other: 'ListeMonoid[T]') -> 'ListeMonoid[T]':
56          return ListeMonoid(self.elements + other.elements)
57
58      @classmethod
59      def empty(cls) -> 'ListeMonoid[T]':
60          return ListeMonoid([])
61
62      def __str__(self):
63          return f"Liste{self.elements}"
64
65  # Fonction generique pour combiner plusieurs elements
66  def combine_all(monoids: list[Monoid[T]]) -> Monoid[T]:
67      if not monoids:
68          return type(monoids[0]).empty() if monoids else None
69      result = monoids[0]
70      for m in monoids[1:]:
71          result = result.combine(m)
72      return result
73
74  # Utilisation
75  nombres_somme = [Somme(1), Somme(2), Somme(3), Somme(4)]
76  resultat_somme = combine_all(nombres_somme)
77  print(f"Somme␣combinee:␣{resultat_somme}")
78
79  nombres_produit = [Produit(2), Produit(3), Produit(4)]
80  resultat_produit = combine_all(nombres_produit)
81  print(f"Produit␣combine:␣{resultat_produit}")
82
```

```python
listes = [ListeMonoid([1, 2]), ListeMonoid([3, 4]), ListeMonoid([5, 6])]
resultat_liste = combine_all(listes)
print(f"Listes combinees: {resultat_liste}")

# Monoid pour les strings
class Concat(Monoid[str]):
    def __init__(self, valeur: str):
        self.valeur = valeur

    def combine(self, other: 'Concat') -> 'Concat':
        return Concat(self.valeur + other.valeur)

    @classmethod
    def empty(cls) -> 'Concat':
        return Concat("")

    def __str__(self):
        return f"Concat('{self.valeur}')"

mots = [Concat("Hello"), Concat(" "), Concat("World"), Concat("!")]
resultat_concat = combine_all(mots)
print(f"Concatenation: {resultat_concat}")
```

Listing 19 – Structures algébriques en Python

9.2 Exemple en OCaml

```
(* Signature pour un Magma *)
module type MAGMA = sig
    type t
    val combine : t -> t -> t
end

(* Signature pour un Semigroup - meme que Magma mais avec
   associativite *)
module type SEMIGROUP = MAGMA

(* Signature pour un Monoid *)
module type MONOID = sig
    include SEMIGROUP
    val empty : t
end

(* Implementation pour l'addition *)
module Somme : MONOID with type t = int = struct
    type t = int
    let combine = ( + )
    let empty = 0
end

```

```
(* Implementation pour la multiplication *)
module Produit : MONOID with type t = int = struct
    type t = int
    let combine = ( * )
    let empty = 1
end

(* Implementation pour la concatenation de listes *)
module ListeConcat (T : sig type t end) : MONOID with type t = T.t
   list = struct
    type t = T.t list
    let combine = List.append
    let empty = []
end

(* Implementation pour la concatenation de strings *)
module StringConcat : MONOID with type t = string = struct
    type t = string
    let combine = ( ^ )
    let empty = ""
end

(* Fonction generique pour combiner plusieurs elements *)
let combine_all (type a) (module M : MONOID with type t = a)
   elements =
    List.fold_left M.combine M.empty elements

(* Utilisation *)
let nombres_somme = [1; 2; 3; 4]
let resultat_somme = combine_all (module Somme) nombres_somme

let nombres_produit = [2; 3; 4]
let resultat_produit = combine_all (module Produit) nombres_produit

let mots = ["Hello"; "␣"; "World"; "!"]
let resultat_concat = combine_all (module StringConcat) mots

let () =
    Printf.printf "Somme␣combinee:␣%d\n" resultat_somme;
    Printf.printf "Produit␣combine:␣%d\n" resultat_produit;
    Printf.printf "Concatenation:␣%s\n" resultat_concat

(* Monoid pour les options *)
module OptionMonoid (M : MONOID) : MONOID with type t = M.t option
   = struct
    type t = M.t option

    let empty = None

    let combine o1 o2 =
        match o1, o2 with
```

```
        | None, None -> None
        | Some x, None -> Some x
        | None, Some y -> Some y
        | Some x, Some y -> Some (M.combine x y)
end

(* Utilisation avec les options *)
module OptionSomme = OptionMonoid(Somme)

let options_somme = [Some 1; None; Some 2; Some 3; None]
let resultat_option_somme = combine_all (module OptionSomme)
   options_somme

let () =
    match resultat_option_somme with
    | Some x -> Printf.printf "Somme des options : %d\n" x
    | None -> print_endline "Aucune valeur dans les options"
```

Listing 20 – Structures algébriques en OCaml

10 Foncteurs (Functors)

10.1 Exemple en Python

```
from typing import TypeVar, Generic, Callable
from abc import ABC, abstractmethod

T = TypeVar('T')
U = TypeVar('U')

class Functor(Generic[T], ABC):
    @abstractmethod
    def map(self, f: Callable[[T], U]) -> 'Functor[U]': pass

# Functor pour Option
class Option(Functor[T], ABC):
    @abstractmethod
    def is_defined(self) -> bool: pass

    @abstractmethod
    def get_or_else(self, default: T) -> T: pass

class Some(Option[T]):
    def __init__(self, value: T):
        self.value = value

    def map(self, f: Callable[[T], U]) -> Option[U]:
        return Some(f(self.value))

    def is_defined(self) -> bool:
```

```
        return True

    def get_or_else(self, default: T) -> T:
        return self.value

    def __str__(self):
        return f"Some({self.value})"

class Nothing(Option[T]):
    def map(self, f: Callable[[T], U]) -> Option[U]:
        return Nothing()

    def is_defined(self) -> bool:
        return False

    def get_or_else(self, default: T) -> T:
        return default

    def __str__(self):
        return "Nothing"

# Functor pour List
class FList(Functor[T]):
    def __init__(self, elements: list[T]):
        self.elements = elements

    def map(self, f: Callable[[T], U]) -> 'FList[U]':
        return FList([f(x) for x in self.elements])

    def __str__(self):
        return f"FList{self.elements}"

# Functor pour Either
class Either(Functor[R], Generic[L, R], ABC):
    @abstractmethod
    def is_left(self) -> bool: pass

    @abstractmethod
    def is_right(self) -> bool: pass

class Left(Either[L, R]):
    def __init__(self, value: L):
        self.value = value

    def map(self, f: Callable[[R], U]) -> Either[L, U]:
        return Left(self.value)

    def is_left(self) -> bool: return True
    def is_right(self) -> bool: return False
    def __str__(self): return f"Left({self.value})"

```

```python
class Right(Either[L, R]):
    def __init__(self, value: R):
        self.value = value

    def map(self, f: Callable[[R], U]) -> Either[L, U]:
        return Right(f(self.value))

    def is_left(self) -> bool: return False
    def is_right(self) -> bool: return True
    def __str__(self): return f"Right({self.value})"

# Loi des foncteurs: preservation de l'identite et de la composition
def test_loi_identite(functor: Functor[T]) -> bool:
    """fmap id = id"""
    identity = lambda x: x
    result = functor.map(identity)
    return str(result) == str(functor)

def test_loi_composition(functor: Functor[T], f: Callable, g: Callable) -> bool:
    """fmap (f . g) = fmap f . fmap g"""
    composition = lambda x: f(g(x))
    result1 = functor.map(composition)
    result2 = functor.map(g).map(f)
    return str(result1) == str(result2)

# Utilisation
option = Some(5)
print(f"Option originale: {option}")
print(f"Option map (*2): {option.map(lambda x: x * 2)}")

flist = FList([1, 2, 3, 4, 5])
print(f"FList originale: {flist}")
print(f"FList map (+1): {flist.map(lambda x: x + 1)}")

either = Right(10)
print(f"Either originale: {either}")
print(f"Either map (/2): {either.map(lambda x: x / 2)}")

# Tests des lois
print(f"Loi identite Option: {test_loi_identite(option)}")
print(f"Loi identite FList: {test_loi_identite(flist)}")

f = lambda x: x + 1
g = lambda x: x * 2
print(f"Loi composition Option: {test_loi_composition(option, f, g)}")
print(f"Loi composition FList: {test_loi_composition(flist, f, g)}")
```

Listing 21 – Foncteurs en Python

10.2 Exemple en OCaml

```
(* Signature d'un Foncteur *)
module type FUNCTOR = sig
    type 'a t
    val map : ('a -> 'b) -> 'a t -> 'b t
end

(* Foncteur pour Option *)
module OptionFunctor : FUNCTOR with type 'a t = 'a option = struct
    type 'a t = 'a option
    let map f = function
        | Some x -> Some (f x)
        | None -> None
end

(* Foncteur pour List *)
module ListFunctor : FUNCTOR with type 'a t = 'a list = struct
    type 'a t = 'a list
    let map = List.map
end

(* Foncteur pour Either *)
module EitherFunctor (L : sig type t end) : FUNCTOR with type 'a t
   = (L.t, 'a) either = struct
    type 'a t = (L.t, 'a) either

    let map f = function
        | Left l -> Left l
        | Right r -> Right (f r)
end

(* Utilisation des foncteurs *)
let option_exemple =
    let open OptionFunctor in
    let option_value = Some 5 in
    map (fun x -> x * 2) option_value

let list_exemple =
    let open ListFunctor in
    let list_value = [1; 2; 3; 4; 5] in
    map (fun x -> x + 1) list_value

module StringEither = EitherFunctor(struct type t = string end)

let either_exemple =
    let either_value = Right 10 in
```

```
45      StringEither.map (fun x -> x / 2) either_value
46
47  let () =
48      Printf.printf "Option map: ";
49      (match option_exemple with
50      | Some x -> Printf.printf "Some %d\n" x
51      | None -> print_endline "None");
52
53      Printf.printf "List map: [";
54      List.iter (Printf.printf "%d; ") list_exemple;
55      Printf.printf "]\n";
56
57      Printf.printf "Either map: ";
58      (match either_exemple with
59      | Left s -> Printf.printf "Left %s\n" s
60      | Right x -> Printf.printf "Right %d\n" x)
61
62  (* Foncteur personnalise pour les arbres binaires *)
63  module TreeFunctor : FUNCTOR with type 'a t = 'a tree = struct
64      type 'a tree =
65          | Leaf
66          | Node of 'a tree * 'a * 'a tree
67
68      let rec map f = function
69          | Leaf -> Leaf
70          | Node (left, value, right) -> Node (map f left, f value,
                map f right)
71  end
72
73  (* Utilisation du foncteur d'arbre *)
74  let arbre_exemple =
75      let open TreeFunctor in
76      let arbre = Node (Node (Leaf, 1, Leaf), 2, Node (Leaf, 3, Leaf)
           ) in
77      map (fun x -> x * 2) arbre
78
79  (* Fonction pour afficher un arbre *)
80  let rec print_tree print_val = function
81      | Leaf -> print_string "Leaf"
82      | Node (left, value, right) ->
83          Printf.printf "Node(";
84          print_tree print_val left;
85          Printf.printf ", ";
86          print_val value;
87          Printf.printf ", ";
88          print_tree print_val right;
89          Printf.printf ")"
90
91  let () =
92      print_string "Arbre map: ";
93      print_tree (Printf.printf "%d") arbre_exemple;
```

```
    print_newline ()

(* Composition de foncteurs *)
module Compose (F : FUNCTOR) (G : FUNCTOR) : FUNCTOR with type 'a t
    = 'a G.t F.t = struct
    type 'a t = 'a G.t F.t

    let map f = F.map (G.map f)
end

(* Composition Option de List *)
module OptionList = Compose(OptionFunctor)(ListFunctor)

let option_list_exemple =
    let value = Some [1; 2; 3] in
    OptionList.map (fun x -> x * 2) value

let () =
    Printf.printf "Option[List] map: ";
    (match option_list_exemple with
    | Some lst ->
        Printf.printf "Some [";
        List.iter (Printf.printf "%d; ") lst;
        Printf.printf "]\n"
    | None -> print_endline "None")
```

Listing 22 – Foncteurs en OCaml

11 Effets de Bord et Monades IO/Task

11.1 Exemple en Python

```
from typing import TypeVar, Generic, Callable, Any
from abc import ABC, abstractmethod
import asyncio

T = TypeVar('T')
U = TypeVar('U')

# Monade IO pour les effets synchrones
class IO(Generic[T], ABC):
    @abstractmethod
    def flat_map(self, f: Callable[[T], 'IO[U]']) -> 'IO[U]': pass

    @abstractmethod
    def run(self) -> T: pass

    def map(self, f: Callable[[T], U]) -> 'IO[U]':
        return self.flat_map(lambda x: IO.pure(f(x)))

```

```python
class IOPure(IO[T]):
    def __init__(self, value: T):
        self.value = value

    def flat_map(self, f: Callable[[T], IO[U]]) -> IO[U]:
        return f(self.value)

    def run(self) -> T:
        return self.value

    def __str__(self):
        return f"IOPure({self.value})"

class IOEffect(IO[T]):
    def __init__(self, effect: Callable[[], T]):
        self.effect = effect

    def flat_map(self, f: Callable[[T], IO[U]]) -> IO[U]:
        return IOEffect(lambda: f(self.effect()).run())

    def run(self) -> T:
        return self.effect()

    def __str__(self):
        return "IOEffect(...)"

class IO:
    @staticmethod
    def pure(value: T) -> IO[T]:
        return IOPure(value)

    @staticmethod
    def effect(effect: Callable[[], T]) -> IO[T]:
        return IOEffect(effect)

# Monade Task pour les effets asynchrones
class Task(Generic[T], ABC):
    @abstractmethod
    async def flat_map(self, f: Callable[[T], 'Task[U]']) -> 'Task[
        U]': pass

    @abstractmethod
    async def run(self) -> T: pass

    async def map(self, f: Callable[[T], U]) -> 'Task[U]':
        return await self.flat_map(lambda x: Task.pure(f(x)))

class TaskPure(Task[T]):
    def __init__(self, value: T):
        self.value = value

```

```
    async def flat_map(self, f: Callable[[T], Task[U]]) -> Task[U]:
        return await f(self.value)

    async def run(self) -> T:
        return self.value

class TaskEffect(Task[T]):
    def __init__(self, effect: Callable[[], Any]):
        self.effect = effect

    async def flat_map(self, f: Callable[[T], Task[U]]) -> Task[U]:
        result = await self.run()
        return await f(result)

    async def run(self) -> T:
        # Simuler une operation asynchrone
        await asyncio.sleep(0.1)
        return self.effect()

class Task:
    @staticmethod
    def pure(value: T) -> Task[T]:
        return TaskPure(value)

    @staticmethod
    def effect(effect: Callable[[], Any]) -> Task[T]:
        return TaskEffect(effect)

# Utilisation de IO
def programme_io() -> IO[str]:
    return (IO.effect(lambda: input("Entrez votre nom: "))
            .flat_map(lambda nom:
                IO.effect(lambda: print(f"Bonjour, {nom}!"))
                .flat_map(lambda _:
                    IO.pure(f"Nom traite: {nom}"))))

# Utilisation de Task
async def programme_task() -> Task[str]:
    return await (Task.effect(lambda: "Donnee recuperee")
            .flat_map(lambda data:
                Task.effect(lambda: print(f"Donnee: {data}"))
                .flat_map(lambda _:
                    Task.pure(f"Donnee traitee: {data}"))))

# Programme principal
if __name__ == "__main__":
    # Execution IO
    print("=== Programme IO ===")
    resultat_io = programme_io().run()
    print(f"Resultat IO: {resultat_io}")

```

```python
    # Execution Task
    print("\n=== Programme Task ===")
    async def run_task():
        resultat_task = await programme_task().run()
        print(f"Resultat Task: {resultat_task}")

    asyncio.run(run_task())

# Exemple plus complexe avec composition
class Config:
    def __init__(self, db_url: str, api_key: str):
        self.db_url = db_url
        self.api_key = api_key

def lire_config() -> IO[Config]:
    return IO.effect(lambda: Config("postgresql://localhost:5432",
        "secret123"))

def connecter_db(config: Config) -> IO[str]:
    return IO.effect(lambda: f"Connecte a {config.db_url}")

def appel_api(config: Config) -> IO[str]:
    return IO.effect(lambda: f"API appelee avec cle: {config.
        api_key}")

def programme_complet() -> IO[str]:
    return (lire_config()
            .flat_map(lambda config:
                connecter_db(config)
                .flat_map(lambda db_msg:
                    appel_api(config)
                    .map(lambda api_msg:
                        f"{db_msg}, {api_msg}"))))

# Execution du programme complet
print("\n=== Programme Complet ===")
resultat_complet = programme_complet().run()
print(f"Resultat complet: {resultat_complet}")
```

Listing 23 – Gestion des effets de bord en Python

11.2 Exemple en OCaml

```ocaml
(* Monade IO pour les effets synchrones *)
module type IO = sig
    type 'a t
    val return : 'a -> 'a t
    val bind : 'a t -> ('a -> 'b t) -> 'b t
    val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
    val run : 'a t -> 'a
end
```

```
9
10 module Io : IO = struct
11     type 'a t = unit -> 'a  (* Les effets sont des thunks *)
12
13     let return x = fun () -> x
14     let bind m f = fun () -> f (m ()) ()
15     let ( >>= ) = bind
16     let run m = m ()
17 end
18
19 (* Fonctions d'effet *)
20 let lire_entree () =
21     print_string "Entrez votre nom: ";
22     read_line ()
23
24 let ecrire_sortie msg =
25     print_endline msg
26
27 (* Programme IO *)
28 let programme_io =
29     Io.return ()
30     >>= fun () -> Io.return (lire_entree ())
31     >>= fun nom ->
32         Io.return (ecrire_sortie ("Bonjour, " ^ nom ^ "!"))
33     >>= fun () ->
34         Io.return ("Nom traite: " ^ nom)
35
36 (* Utilisation *)
37 let () =
38     print_endline "=== Programme IO ===";
39     let resultat = Io.run programme_io in
40     Printf.printf "Resultat: %s\n" resultat
41
42 (* Monade Lazy pour l'evaluation paresseuse *)
43 module type Lazy = sig
44     type 'a t
45     val delay : (unit -> 'a) -> 'a t
46     val force : 'a t -> 'a
47 end
48
49 module Lazy : Lazy = struct
50     type 'a t = 'a option ref * (unit -> 'a)
51
52     let delay f = (ref None, f)
53
54     let force (cache, f) =
55         match !cache with
56         | Some x -> x
57         | None ->
58             let x = f () in
59             cache := Some x;
```

```
            x
end

(* Exemple d'evaluation paresseuse *)
let calcul_lourd () =
    print_endline "Calcul en cours...";
    let rec fib n = if n <= 1 then n else fib (n - 1) + fib (n - 2)
        in
    fib 10

let valeur_paresseuse = Lazy.delay calcul_lourd

let () =
    print_endline "\n=== Evaluation Paresseuse ===";
    print_endline "Premier appel:";
    let result1 = Lazy.force valeur_paresseuse in
    Printf.printf "Resultat: %d\n" result1;
    print_endline "Deuxieme appel (deja calcule):";
    let result2 = Lazy.force valeur_paresseuse in
    Printf.printf "Resultat: %d\n" result2

(* Gestion d'erreurs avec Result *)
let diviser a b =
    if b = 0 then Error "Division par zero"
    else Ok (a / b)

let calcul_complexe x y z =
    diviser x y
    |> Result.bind (fun r1 -> diviser r1 z)
    |> Result.map (fun r2 -> r2 * 2)

let () =
    print_endline "\n=== Gestion d'Erreurs ===";
    match calcul_complexe 10 2 2 with
    | Ok result -> Printf.printf "Succes: %d\n" result
    | Error msg -> Printf.printf "Erreur: %s\n" msg;

    match calcul_complexe 10 0 2 with
    | Ok result -> Printf.printf "Succes: %d\n" result
    | Error msg -> Printf.printf "Erreur: %s\n" msg

(* Composition de monades *)
module type MONAD = sig
    type 'a t
    val return : 'a -> 'a t
    val bind : 'a t -> ('a -> 'b t) -> 'b t
end

(* Transformateur de monade Option *)
module OptionT (M : MONAD) : MONAD with type 'a t = 'a option M.t =
    struct
```

```
109     type 'a t = 'a option M.t
110
111     let return x = M.return (Some x)
112
113     let bind m f =
114         M.bind m (function
115             | None -> M.return None
116             | Some x -> f x)
117 end
118
119 (* Exemple avec OptionT *)
120 module IoMonad : MONAD with type 'a t = 'a Io.t = struct
121     type 'a t = 'a Io.t
122     let return = Io.return
123     let bind = Io.bind
124 end
125
126 module IoOption = OptionT(IoMonad)
127
128 let programme_io_option =
129     IoOption.return 5
130     >>= fun x -> IoOption.return (x * 2)
131     >>= fun y -> if y > 10 then IoOption.return y else IoMonad.
        return None
132
133 let () =
134     print_endline "\n=== OptionT IO ===";
135     match Io.run programme_io_option with
136     | Some x -> Printf.printf "Resultat: %d\n" x
137     | None -> print_endline "Aucun resultat"
```

Listing 24 – Gestion des effets de bord en OCaml

12 Conclusion

Ce document a couvert les concepts fondamentaux de la programmation fonctionnelle avec des implémentations en Python et OCaml. Les principaux avantages observés sont :

- **OCaml** : Typage fort, pattern matching natif, meilleures performances
- **Python** : Expressivité, riche écosystème, adoption large
- **Concepts communs** : Immutabilité, pureté, composition

La programmation fonctionnelle offre une approche robuste pour construire des applications maintenables et testables.