

Éléments de Programmation

Programmation Fonctionnelle

La programmation fonctionnelle traite le calcul comme l'évaluation de fonctions mathématiques. Elle évite les états mutables et met l'accent sur les **fonctions pures** (même entrée → même sortie, sans effets secondaires).

Expressions

Expressions booléennes

- Constantes : `True`, `False`
- Opérateurs : `not`, `&&` (conjonction), `||` (disjonction)

Comparaisons

- `==` (égalité), `!=` (inégalité)
- `<`, `<=`, `>`, `>=`
- `elem` (appartenance)

⚠ Warning

Dans le document original, `/=` est noté pour “appartenance”, mais en Haskell `/=` signifie “différent de” (not equal). L'appartenance se vérifie avec `elem`.

Expression let

Permet de définir des variables/fonctions locales avant de retourner une valeur.

```
cylinder r h = let
  side = 2 * pi * r * h
  base = pi * r^2
in side + 2 * base
```

Fonctions

Syntaxe : `functionName param1 param2 = expression`

Propriétés des fonctions Haskell :

- Au moins un paramètre
- Retourne toujours une valeur
- Fonctions pures (même entrée → même sortie)

Modèle de substitution

Méthode d'évaluation remplaçant les variables par leurs valeurs, utile pour comprendre l'exécution.

Gardes (Guards)

Permettent de définir des conditions dans les fonctions.

```
abs n | n >= 0 = n
      | otherwise = -n
```

⚠ Warning

Si les gardes ne couvrent pas tous les cas possibles, une erreur d'exécution peut se produire.

Clause where

Définit des variables/fonctions locales à la fin d'une définition.

```
cylinder r h = side + 2 * base
  where
    side = 2 * pi * r * h
    base = pi * r^2
```

Listes

Collections ordonnées d'éléments **homogènes** (même type).

Propriétés des listes :

- **Immutables** (non modifiables après création)
- **Récurives** (vide `[]` ou tête + queue)
- **Homogènes** (tous les éléments du même type)

Construction

Opérateur : `(cons)` - associatif à droite

`numbers = 1 : 2 : 3 : []` -- équivaut à `[1,2,3]`

Opérateurs standard

- `length xs` : longueur
- `head xs` : premier élément
- `tail xs` : liste sans premier élément
- `last xs` : dernier élément
- `init xs` : liste sans dernier élément
- `take n xs` : n premiers éléments
- `drop n xs` : liste sans n premiers éléments
- `xs !! n` : n-ième élément (indexé à partir de 0)
- `xs ++ ys` : concatène deux listes
- `elem x xs` : vérifie si x est dans xs
- `reverse xs` : inverse la liste

Listes infinies

Grâce à l'évaluation paresseuse (lazy evaluation).

```
naturals = [0..]
firstTen = take 10 naturals
```

Compréhensions de Listes

Syntaxe inspirée des mathématiques pour générer des listes.

```
squares = [x^2 | x <- [1..10]]
evens = [x | x <- [1..10], even x]
```

Générateurs multiples

```
pairs = [(x,y) | x <- [1,2,3], y <- ['a','b']]
```

Prédicats

Filtrent les éléments générés.

```
evens = [x | x <- [1..20], even x]
```

Déclarations locales

Avec `let` dans la compréhension.

```
[y | x <- ['a'..'z'], let y = toUpper x]
```

Tuples

Collections ordonnées d'éléments de **types différents**.

```
person = (42, "Alice", True)
```

i Info

- Pas de tuple à un seul élément
- **Paires** (tuples à 2 éléments) : `fst` et `snd` pour accéder aux éléments

Pattern Matching

Technique pour décomposer des structures de données.

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Cas par défaut

Utiliser `_` pour tout autre cas

```
vowel 'a' = True
vowel _ = False
```

Avec les listes

```
head (x:_) = x
tail (_:xs) = xs
```

Expression case

```
describeList xs = case xs of
  [] -> "vide"
  [x] -> "un élément"
  (x:y:_) -> "plusieurs éléments"
```

⚠ Warning

Patterns non-exhaustifs → erreur d'exécution

Récursion

Récursivité terminale

Appel récursif = dernière opération (optimisable).

```
sum n = sumAux n 0
  where sumAux n total
    | n <= 0 = total
    | otherwise = sumAux (n-1) (n + total)
```

Récursivité non-terminale

Opérations après l'appel récursif.

```
sum n | n <= 0 = 0
      | otherwise = n + sum (n-1)
```

Système de Types

Caractéristiques du typage Haskell

- **Typage fort** et statique
- **Inférence de types**
- **Polymorphisme paramétrique**
- **Type-safe** : erreurs détectées à la compilation

i Info

Les programmes Haskell peuvent être considérés comme **type-safe** (sécurité de type). Les erreurs de types sont détectées à la compilation et non à l'exécution.

Typage d'Expressions

Syntaxe : `expression :: Type`

```
'c' :: Char
[1,2,3] :: [Int]
(1, 'a', True) :: (Int, Char, Bool)
```

Fonctions

```
even :: Int -> Bool
gcd  :: Int -> Int -> Int  -- prend deux Int, retourne un Int
```

Signature de type

Écrire avant la définition, améliore la lisibilité.

```
even :: Int -> Bool
even n = n `mod` 2 == 0
```

i Info

En Haskell, il est recommandé de toujours fournir une signature de type explicite pour les fonctions de haut niveau (top-level functions).

Polymorphisme

Polymorphisme paramétrique

Fonctions fonctionnant avec n’importe quel type.

```
length :: [a] -> Int  -- 'a' est un type générique
length [] = 0
length (_,xs) = 1 + length xs
```

Polymorphisme ad-hoc

Via les classes de types - comportements différents selon les types.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Fonction partielle

Une fonction partielle est une fonction qui n’est pas définie pour tous les arguments possibles.

```
head :: [a] -> a
head (x:_) = x  -- erreur si liste vide
```

Classes de types

Les classes de types en Haskell sont un mécanisme qui permet de définir des interfaces pour les types.

```
(==) :: Eq a => a -> a -> Bool  -- doit être Eq
(+)  :: Num a => a -> a -> a    -- doit être Num
show :: Show a => a -> String  -- doit être Show
```

Fonctions d’Ordre Supérieur

Introduction

Les fonctions d’ordre supérieur (Higher-Order Functions, HOF) sont des fonctions qui peuvent :

- Prendre d’autres fonctions en argument, OU
- Retourner des fonctions

```
sum f a b
  | a > b = 0
  | otherwise = f a + sum f (a+1) b
```

Expressions Lambda

Fonctions anonymes.

```
\x -> x * x * x  -- fonction cube
```

La notation \x indique que nous définissons une fonction prenant un argument x.

Fermetures (Closures)

Une fermeture est une fonction qui capture les variables de son environnement lexical.

```
makeMultiplier factor = \x -> x * factor
double = makeMultiplier 2
triple  = makeMultiplier 3

result1 = double 5  -- 10
result2 = triple 5  -- 15
```

Currying (Fonctions curriées)

Le currying est une technique de transformation de fonctions qui permet de convertir une fonction prenant plusieurs arguments en une série de fonctions prenant un seul argument chacune. En Haskell, **toutes les fonctions sont curriées par défaut**.

```
add :: Int -> Int -> Int
add x y = x + y
-- équivaut à
add x = \y -> x + y
```

i Info

Le principe de la currification consiste à transformer une fonction prenant un tuple en argument en une fonction prenant le premier élément du tuple et retournant une fonction prenant le second élément du tuple.

- curry : transforme une fonction prenant un tuple en une fonction curriée
- uncurry : transforme une fonction curriée en une fonction prenant un tuple

Application partielle

L’application partielle est le processus de fixation d’un certain nombre d’arguments d’une fonction pour produire une nouvelle fonction avec un nombre réduit d’arguments.

Application operator

L’opérateur \$ est utilisé pour appliquer une fonction à un argument, en évitant la nécessité de parenthèses.

```
f $ g x  -- au lieu de f (g x)
```

Flip operator

L’opérateur flip prend une fonction à deux arguments et retourne une nouvelle fonction avec les arguments inversés.

List processing

Map

Applique une fonction à chaque élément d’une liste.

```
map :: (a -> b) -> [a] -> [b]
```

Filter

Filtre les éléments d’une liste en fonction d’une condition donnée.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]

filter even [1..10]  -- [2,4,6,8,10]
filter (>5) [1..10]  -- [6,7,8,9,10]
```

Folding (Réduction)

Le folding est une technique de réduction d’une liste à une seule valeur en appliquant une fonction binaire de manière récursive.

Fold right


```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Commence à la fin de la liste et applique la fonction binaire de droite à gauche.

Fold left

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Commence au début de la liste et applique la fonction binaire de gauche à droite.

 Warning

En fonction de notre besoin, nous choisirons foldl ou foldr. Par exemple, pour élever une liste de nombres à une puissance, nous utiliserons foldl car l’opération d’exponentiation n’est pas associative.

Le cas de la soustraction est plus délicat car elle n’est ni associative ni commutative :

- foldl (-) 0 [1,2,3] donne 0 - 1 - 2 - 3 = -6
- foldr (-) 0 [1,2,3] donne 1 - (2 - (3 - 0)) = 2

Composition de fonctions

La composition de fonctions permet de combiner plusieurs fonctions en une seule. L’opérateur (.) est utilisé pour composer deux fonctions.

```
filter (odd . fst) [(65, "A"), (66, "B"), (67, "C")]
-- Résultat : [(65,"A"),(67,"C")]
```

Types de Données Algébriques

Introduction

En Haskell, pour définir des types de données algébriques, on utilise le mot-clé data.

```
data Color = Red | Green | Blue
```

```
rgb Red = (255, 0, 0)
rgb Green = (0, 255, 0)
rgb Blue = (0, 0, 255)
```

i Info

Dans ce genre d’exemple, nous parlons principalement d’enum (énumérations), qui sont des types de données algébriques simples avec plusieurs constructeurs sans arguments.

Types avec paramètres

Les types de données algébriques peuvent également avoir des paramètres.

```
data Person = Person Int String Int
-- ID, nom, âge


> Person 1 "John" 42
```

Pattern Matching

Le pattern matching permet de décomposer les valeurs des types de données algébriques.

```
getName :: Person -> String
getName (Person _ name _) = name

setName :: String -> Person -> Person
setName name (Person id _ age) = Person id name age
```

 **Hint**

Si l'on souhaite pouvoir afficher les données d'un type personnalisé, on peut dériver l'instance Show automatiquement :

```
data Color = Red | Green | Blue deriving (Show)
```

Terminologie

Chaque type algébrique ne peut avoir qu'un seul constructeur de type, mais ce constructeur peut avoir plusieurs formes (constructeurs de données).

Nombre de valeurs possibles :

```
data Bool = True | False -- 1+1 = 2 valeurs
data TwoBools = TwoBools Bool Bool -- 2*2 = 4 valeurs
data Complex = Two Bool Bool | One Bool | None -- 2*2+2+1 = 7 valeurs
```

Types paramétriques

Les types paramétriques permettent de définir des types de données génériques qui peuvent fonctionner avec n'importe quel type de données.

```
data Pair a b = Pair a b
```

```
intStringPair = Pair 1 "one"
floatBoolPair = Pair 3.14 True
```

Alias et nouveaux types

Type alias

Un alias de type permet de donner un nom alternatif à un type existant.

```
type String = [Char]
```

Alias polymorphiques

```
type Pair a b = (a, b)
```


```
first :: Pair a b -> a
first (x, _) = x
```

Nouveaux types

Les nouveaux types permettent de créer des types distincts basés sur des types existants, offrant une meilleure sécurité de type.

```
newtype RGB = RGB (Int, Int, Int) deriving (Show)
```

```
> RGB (255, 0, 0)
```

 **Info**

newtype permet d'offrir une meilleure performance que data lorsqu'il n'y a qu'un seul constructeur avec un seul champ, car il n'introduit pas de surcharge supplémentaire au moment de l'exécution.

Types de données récursifs

Les types de données récursifs sont des types qui se définissent en termes d'eux-mêmes.

```
data List t = Nil | Cons t (List t)
```

```
chars :: List Char
chars = Cons 'a' (Cons 'b' (Cons 'c' Nil))
```

```
bools :: List Bool
bools = Cons True (Cons False Nil)
```

Built-in lists

Haskell fournit un type de liste intégré qui est défini de manière similaire. Les listes en Haskell sont donc des types de données récursifs.

Exemple : Matryoshka

```
data Matryoshka = Hollow | Nesting Matryoshka
```

```
dolls :: Matryoshka -> Int
dolls Hollow = 1
dolls (Nesting m) = 1 + dolls m
```

Maybe et Either

Gestion des erreurs

En Haskell, les types Maybe et Either sont utilisés pour gérer les erreurs et les valeurs optionnelles de manière sûre.

Prenons le cas de la division par zéro :

```
div :: Float -> Float -> Float
div x 0 = error "Division by zero"
```

Maybe

Pour éviter l'utilisation de error, on peut utiliser le type Maybe :

```
safeDiv :: Float -> Float -> Maybe Float
safeDiv x 0 = Nothing
safeDiv x y = Just (x / y)
```

Maybe permet donc d'encapsuler une valeur qui peut être absente, ce qui force le programmeur à gérer explicitement le cas où la valeur n'est pas présente.

Either

Le type Either est une autre façon de gérer les erreurs, en permettant de retourner soit une valeur valide, soit une erreur avec un message.

```
safeDiv :: Int -> Int -> Either String Int
safeDiv x 0 = Left "Division by zero"
safeDiv x y = Right (x `div` y)
```

- Left est utilisé pour représenter une erreur avec un message
- Right représente une valeur valide

Exemple : fonction find

```
find :: (a -> Bool) -> [a] -> Maybe a
find _ [] = Nothing
find p (x:xs)
  | p x = Just x
  | otherwise = find p xs
```

Records

Les records en Haskell sont une extension des types de données algébriques qui permettent de nommer les champs d'un constructeur de données.

```
data Point = Point { x :: Float, y :: Float } deriving (Show)
```

```
a = Point 3 4
b = a { x = 5 } -- Mise à jour du champ x
> b
Point {x = 5.0, y = 4.0}
```

Records pattern matching

Le pattern matching avec les records permet d'extraire facilement les champs nommés.

```
getX :: Point -> Float
getX (Point { x = xCoord }) = xCoord
```

```
getY :: Point -> Float
getY (Point { y = yCoord }) = yCoord
```

Exemple : Language

```
data Paradigm = Functional | Imperative | Object
```

```
data Language = Language {
  name :: String,
  paradigm :: Paradigm
}
```

```
isFunctional :: Language -> Bool
isFunctional Language { paradigm = Functional } = True
isFunctional _ = False
```

Classes de types et instances

Les classes de types en Haskell permettent de définir des interfaces génériques que les types peuvent implémenter.

Deriving

Haskell permet de dériver automatiquement des instances de certaines classes de types.

```
data Shape = Circle Float | Rectangle Float Float
  deriving (Show, Eq)
```

Heritage multiple

Un type de données peut dériver des instances de plusieurs classes de types en même temps en les séparant par des virgules dans la clause deriving.


Définir une classe de types

```
class Comparable a where
  cmp :: a -> a -> Int
```

```
data Bit = Zero | One deriving Show
```


```
instance Comparable Bit where
  cmp One Zero = 1
  cmp Zero One = -1
  cmp _ _ = 0
```

Points Clés à Retenir

 **Info**


Concepts fondamentaux :

- Fonctions pures** : pas d'effets secondaires, même entrée → même sortie
- Immutabilité** : listes et données non modifiables
- Pattern matching** : décomposition de structures, attention aux cas non-exhaustifs
- Récursion** : terminale (optimisable) vs non-terminale

 **Info**

Système de types :

- Types** : fort, statique, inféré, type-safe
- Polymorphisme** : paramétrique (générique) et ad-hoc (classes de types)

 **Info**

Fonctions avancées :

7. **HOF** : fonctions prenant/retournant des fonctions
8. **Currying** : toutes les fonctions sont curryfiées en Haskell
9. **Folding** : `foldr` (droite) vs `foldl` (gauche), attention à l'associativité

i Info**Types de données :**

10. **ADT** : data pour nouveaux types, type pour alias, newtype pour encapsulation
11. **Gestion d'erreurs** : `Maybe` (optionnel) et `Either` (erreur + message)
12. **Records** : accès par nom aux champs