Programmation Fonctionnelle

Illustrée par le langage Haskell

Didier Lime

École Centrale de Nantes – LS2N

Année 2019 – 2020

Plan

Plan

Programmation fonctionnelle

- Décrire le programme comme la **composition** de transformations décrites par des **fonctions** mathématiques ;

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{sinon} \end{cases}$$

Programmation fonctionnelle

- Décrire le programme comme la **composition** de transformations décrites par des **fonctions** mathématiques ;

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{sinon} \end{cases}$$

- Un cas particulier de **programmation déclarative** ;

Programmation fonctionnelle

- Décrire le programme comme la **composition** de transformations décrites par des **fonctions** mathématiques ;

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{sinon} \end{cases}$$

- Un cas particulier de **programmation déclarative** ;
- Basé sur le **λ-calcul** d'Alonzo Church (1935) ;

Programmation fonctionnelle

- Décrire le programme comme la **composition** de transformations décrites par des **fonctions** mathématiques ;

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{sinon} \end{cases}$$

- Un cas particulier de **programmation déclarative** ;
- Basé sur le λ-**calcul** d'Alonzo Church (1935) ;
- Fonctions = valeurs de **première classe** ;

Programmation fonctionnelle

- Décrire le programme comme la **composition** de transformations décrites par des **fonctions** mathématiques ;

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{sinon} \end{cases}$$

- Un cas particulier de **programmation déclarative** ;
- Basé sur le **λ-calcul** d'Alonzo Church (1935) ;
- Fonctions = valeurs de **première classe** ;
- Pas de notion de variable, d'ordre d'évaluation séquentiel, d'effets de bords $\Rightarrow$ Langages fonctionnels « **purs** »

Programmation fonctionnelle

- Décrire le programme comme la **composition** de transformations décrites par des **fonctions** mathématiques ;

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{sinon} \end{cases}$$

- Un cas particulier de **programmation déclarative** ;
- Basé sur le **λ-calcul** d'Alonzo Church (1935) ;
- Fonctions = valeurs de **première classe** ;
- Pas de notion de variable, d'ordre d'évaluation séquentiel, d'effets de bords $\Rightarrow$ Langages fonctionnels « **purs** »
- Exemples (purs et impurs) : Lisp (1958), ML (1973), Scheme (1975), (O)CAML (1985), **Haskell** (1990), Scala (2003), ...

Propriétés : Transparence référentielle

Une **même fonction** *appliquée sur les* **mêmes arguments** *renvoie le* **même résultat**

Propriétés : Transparence référentielle

Une **même fonction** *appliquée sur les* **mêmes arguments** *renvoie le* **même résultat**

- Facilite le raisonnement et la **parallélisation** ;

Propriétés : Transparence référentielle

Une **même fonction** *appliquée sur les* **mêmes arguments** *renvoie le* **même résultat**

- Facilite le raisonnement et la **parallélisation** ;
- Pas de pointeurs ou de variables globales... ou d'entrées-sorties.

Propriétés : Transparence référentielle

Une **même fonction** *appliquée sur les* **mêmes arguments** *renvoie le* **même résultat**

- Facile le raisonnement et la **parallélisation** ;
- Pas de pointeurs ou de variables globales... ou d'entrées-sorties.
- Purement fonctionnel jusqu'à un certain point :
  - autoriser du code impur sous certaines conditions

```
f::Integer->Integer
f 0 = 1
f n = n * f (n-1)

main::IO ()
main = print (f 12)
```

  - **séparer** clairement le code pur et le code impur.

Propriétés : Immutabilité

La valeur des objets manipulés ne peut **jamais** *être* **modifiée**

Propriétés : Immutabilité

La valeur des objets manipulés ne peut **jamais** *être* **modifiée**

- Toute « modification » produit un nouvel objet (p. ex. changer la valeur d'une case d'un tableau)

Propriétés : Immutabilité

La valeur des objets manipulés ne peut **jamais** *être* **modifiée**

- Toute « modification » produit un nouvel objet (p. ex. changer la valeur d'une case d'un tableau)
- Structures de données **purement fonctionnelles** (partage de données) ;

Propriétés : Immutabilité

La valeur des objets manipulés ne peut **jamais** *être* **modifiée**

- Toute « modification » produit un nouvel objet (p. ex. changer la valeur d'une case d'un tableau)
- Structures de données **purement fonctionnelles** (partage de données) ;
- Utilisation **raisonnée** de code impur pour l'efficacité
  P. ex. : abstraction pure compilée vers du code mutable

Le langage Haskell

- Issu d'un **effort commun** de la communauté scientifique autour de la programmation fonctionnelle ;

Le langage Haskell

- Issu d'un **effort commun** de la communauté scientifique autour de la programmation fonctionnelle ;
- Créé en 1990 à partir de trois ans de travaux d'un comité d'experts ;

Le langage Haskell

- Issu d'un **effort commun** de la communauté scientifique autour de la programmation fonctionnelle ;
- Créé en 1990 à partir de trois ans de travaux d'un comité d'experts ;
- Nommé en hommage au logicien Haskell Brooks Curry ;

Le langage Haskell

- Issu d'un **effort commun** de la communauté scientifique autour de la programmation fonctionnelle ;
- Créé en 1990 à partir de trois ans de travaux d'un comité d'experts ;
- Nommé en hommage au logicien Haskell Brooks Curry ;
- Deux révisions principales : Haskell 98 et Haskell 2010 ;

Le langage Haskell

- Issu d'un **effort commun** de la communauté scientifique autour de la programmation fonctionnelle ;
- Créé en 1990 à partir de trois ans de travaux d'un comité d'experts ;
- Nommé en hommage au logicien Haskell Brooks Curry ;
- Deux révisions principales : Haskell 98 et Haskell 2010 ;
- Caractéristiques :
  - langage fonctionnel **pur** ;
  - basé sur le lambda calcul typé ;
  - **fortement** et **statiquement** typé, polymorphique ;
  - évaluation **non-stricte** ;
  - syntaxe proche des mathématiques.

Le langage Haskell

- Issu d'un **effort commun** de la communauté scientifique autour de la programmation fonctionnelle ;
- Créé en 1990 à partir de trois ans de travaux d'un comité d'experts ;
- Nommé en hommage au logicien Haskell Brooks Curry ;
- Deux révisions principales : Haskell 98 et Haskell 2010 ;
- Caractéristiques :
  - langage fonctionnel **pur** ;
  - basé sur le lambda calcul typé ;
  - **fortement** et **statiquement** typé, polymorphique ;
  - évaluation **non-stricte** ;
  - syntaxe proche des mathématiques.
- Multiples compilateurs et interpréteurs : le standard *de facto* est le *Glasgow Haskell Compiler* (**GHC**).

Plan

Composition

Si $f : A \to B$ *et* $g : B \to C$, *alors* $g \circ f : A \to C$ *et*

$$\forall x \in A, (g \circ f)(x) = g(f(x))$$

- Construire des programmes **complexes** à partir de fonctions **simples** ;

Composition

Si $f : A \to B$ *et* $g : B \to C$, *alors* $g \circ f : A \to C$ *et*

$$\forall x \in A, (g \circ f)(x) = g(f(x))$$

- Construire des programmes **complexes** à partir de fonctions **simples** ;
- Utilisation implicite :

```
-- Les noms de fonctions et variables
-- commencent par une minuscule en Haskell
square x = x*x
inc x = x+1

f x = inc (square x)
```

Composition

Si $f : A \to B$ *et* $g : B \to C$*, alors* $g \circ f : A \to C$ *et*

$$\forall x \in A, (g \circ f)(x) = g(f(x))$$

- Construire des programmes **complexes** à partir de fonctions **simples** ;
- Utilisation implicite :

```
-- Les noms de fonctions et variables
-- commencent par une minuscule en Haskell
 square x = x*x
 inc x = x+1

 f x = inc (square x)
```

- ou explicite :

```
f x = (inc.square) x -- ou juste f = inc.square
```

Composition

Si $f : A \to B$ *et* $g : B \to C$*, alors* $g \circ f : A \to C$ *et*

$$\forall x \in A, (g \circ f)(x) = g(f(x))$$

- Construire des programmes **complexes** à partir de fonctions **simples** ;
- Utilisation implicite :

```haskell
-- Les noms de fonctions et variables
-- commencent par une minuscule en Haskell
 square x = x*x
 inc x = x+1

 f x = inc (square x)
```

- ou explicite :

```haskell
f x = (inc.square) x -- ou juste f = inc.square
```

- L'**élement neutre** pour la composition est la fonction **identité** :

```haskell
f.id == id.f == f -- id est la fonction identité
```

Évaluation conditionnelle

- La valeur d'une fonction peut-être définie de façon **conditionnelle** :

```
    f n = if n == 0 then 1 else n * f (n-1)
```

- Remarques :
  - Ce `if` est une **expression** ;
  - Définition ($\stackrel{\text{def}}{=}$) avec `=` ;
  - Égalité booléenne avec `==` ;
  - Différence booléenne ($\neq$) avec `/=`.

Évaluation conditionnelle

- La valeur d'une fonction peut-être définie de façon **conditionnelle** :

```
f n = if n == 0 then 1 else n * f (n-1)
```

- Remarques :
  - Ce `if` est une **expression** ;
  - Définition ($\stackrel{\text{def}}{=}$) avec = ;
  - Égalité booléenne avec == ;
  - Différence booléenne ($\neq$) avec /=.
- Évaluation **gardée** :

```
signe n
    | n<0       = -1
    | n>0       = 1
    | otherwise = 0
```

Évaluation conditionnelle

- Filtrage par motif (***Pattern matching***) :

```haskell
-- le plus idiomatique
f 0 = 1
f n =  n * f (n-1)

-- ou avec case ... of pour avoir une expression
f n = case n of
    0 -> 1
    _ -> n * f (n-1) -- _ correspond à tout

-- Fonction constante:
trois _ = 3
```

Récursivité

Une fonction est **récursive** *si elle est définie en fonction d'elle-même*

Récursivité

Une fonction est **récursive** *si elle est définie en fonction d'elle-même*

- Pas de structure de **répétition** native dans la programmation fonctionnelle ;

Récursivité

Une fonction est **récursive** *si elle est définie en fonction d'elle-même*

- Pas de structure de **répétition** native dans la programmation fonctionnelle ;
- On peut utiliser la **récursivité** pour accomplir des tâches similaires ;

```
f 0 = 1
f n =  n * f (n-1)
```

Récursivité

Une fonction est **récursive** *si elle est définie en fonction d'elle-même*

- Pas de structure de **répétition** native dans la programmation fonctionnelle ;
- On peut utiliser la **récursivité** pour accomplir des tâches similaires ;

```
f 0 = 1
f n =  n * f (n-1)
```

- Une fonction récursive possède :
  - un (ou plusieurs) cas général récursif ;
  - un (ou plusieurs) cas de base **non** récursif.

Récursivité

Une fonction est **récursive** *si elle est définie en fonction d'elle-même*

- Pas de structure de **répétition** native dans la programmation fonctionnelle ;
- On peut utiliser la **récursivité** pour accomplir des tâches similaires ;

```
f 0 = 1
f n =  n * f (n-1)
```

- Une fonction récursive possède :
  - un (ou plusieurs) cas général récursif ;
  - un (ou plusieurs) cas de base **non** récursif.
- Une fonction récursive n'est **bien définie** que si pour toute valeur du domaine, les cas récursifs ne sont évalués qu'un nombre **fini** de fois.

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

$3 * ??$

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

2 * ??
3 * ??

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

1 * ??
2 * ??
3 * ??

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

1
1 * ??
2 * ??
3 * ??

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

1 * **1**
2 * ??
3 * ??

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

1
2 * ??
3 * ??

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```haskell
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

2 * **1**

3 * ??

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

2
3 * ??

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```haskell
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

$3 * \mathbf{2}$

Évaluation d'une fonction récursive

- L'évaluation d'une fonction récursive se fait sur le modèle d'une **pile** :

```
f 0 = 1
f n =  n * f (n-1)

main = print (f 3)
```

6

Fonction récursive terminale

Pour une fonction f définie en utilisant une fonction g, l'évaluation de g est dite **terminale** *(tail evaluation/call) si elle fournit directement la valeur de f sans opération supplémentaire.*

Fonction récursive terminale

Pour une fonction f définie en utilisant une fonction g, l'évaluation de g est dite **terminale** *(tail evaluation/call) si elle fournit directement la valeur de f sans opération supplémentaire.*

- Une fonction **récursive** est dite **terminale** si tous les cas récursifs sont des évaluations terminales ;

Fonction récursive terminale

Pour une fonction f définie en utilisant une fonction g, l'évaluation de g est dite **terminale** *(tail evaluation/call) si elle fournit directement la valeur de f sans opération supplémentaire.*

- Une fonction **récursive** est dite **terminale** si tous les cas récursifs sont des évaluations terminales ;
- Une évaluation terminale ne nécessite pas la mémoire de résultats intermédiaires de la fonction « externe » ;

Fonction récursive terminale

Pour une fonction f définie en utilisant une fonction g, l'évaluation de g est dite **terminale** *(tail evaluation/call) si elle fournit directement la valeur de f sans opération supplémentaire.*

- Une fonction **récursive** est dite **terminale** si tous les cas récursifs sont des évaluations terminales ;
- Une évaluation terminale ne nécessite pas la mémoire de résultats intermédiaires de la fonction « externe » ;
- On peut donc les **évaluer** plus efficacement :

Fonction récursive terminale

Pour une fonction f définie en utilisant une fonction g, l'évaluation de g est dite **terminale** *(tail evaluation/call) si elle fournit directement la valeur de f sans opération supplémentaire.*

- Une fonction **récursive** est dite **terminale** si tous les cas récursifs sont des évaluations terminales ;
- Une évaluation terminale ne nécessite pas la mémoire de résultats intermédiaires de la fonction « externe » ;
- On peut donc les **évaluer** plus efficacement :

```
f' 0 r = r
f' n r = f' (n-1) (r*n)

f n = f' n 1

main = print (f 3)
```

Fonction récursive terminale

Pour une fonction f définie en utilisant une fonction g, l'évaluation de g est dite **terminale** *(tail evaluation/call) si elle fournit directement la valeur de f sans opération supplémentaire.*

- Une fonction **récursive** est dite **terminale** si tous les cas récursifs sont des évaluations terminales ;
- Une évaluation terminale ne nécessite pas la mémoire de résultats intermédiaires de la fonction « externe » ;
- On peut donc les **évaluer** plus efficacement :

```
f' 0 r = r
f' n r = f' (n-1) (r*n)

f n = f' n 1

main = print (f 3)
```

n=3,r=1

Fonction récursive terminale

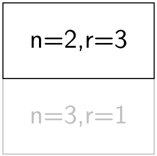
Pour une fonction f définie en utilisant une fonction g, l'évaluation de g est dite **terminale** *(tail evaluation/call) si elle fournit directement la valeur de f sans opération supplémentaire.*

- Une fonction **récursive** est dite **terminale** si tous les cas récursifs sont des évaluations terminales ;
- Une évaluation terminale ne nécessite pas la mémoire de résultats intermédiaires de la fonction « externe » ;
- On peut donc les **évaluer** plus efficacement :

```
f' 0 r = r
f' n r = f' (n-1) (r*n)

f n = f' n 1

main = print (f 3)
```



Fonction récursive terminale

Pour une fonction f définie en utilisant une fonction g, l'évaluation de g est dite **terminale** *(tail evaluation/call) si elle fournit directement la valeur de f sans opération supplémentaire.*

- Une fonction **récursive** est dite **terminale** si tous les cas récursifs sont des évaluations terminales ;
- Une évaluation terminale ne nécessite pas la mémoire de résultats intermédiaires de la fonction « externe » ;
- On peut donc les **évaluer** plus efficacement :

```
f' 0 r = r
f' n r = f' (n-1) (r*n)

f n = f' n 1

main = print (f 3)
```

n=1,r=6
n=2,r=3
n=3,r=1

Fonction récursive terminale

Pour une fonction f définie en utilisant une fonction g, l'évaluation de g est dite **terminale** *(tail evaluation/call) si elle fournit directement la valeur de f sans opération supplémentaire.*

- Une fonction **récursive** est dite **terminale** si tous les cas récursifs sont des évaluations terminales ;
- Une évaluation terminale ne nécessite pas la mémoire de résultats intermédiaires de la fonction « externe » ;
- On peut donc les **évaluer** plus efficacement :

```
f' 0 r = r
f' n r = f' (n-1) (r*n)

f n = f' n 1

main = print (f 3)
```

n=0,r=**6**
n=1,r=6
n=2,r=3
n=3,r=1

Récursivité : Exercices

Exercice

1. Écrire une fonction récursive qui calcule la puissance n^{e} d'un nombre ;
2. Utiliser le fait que $x^{2n} = (x^2)^n$ pour améliorer la fonction.
3. Quelles sont les complexités temporelles des deux versions ?

Récursivité : Exercices

Exercice

1. Écrire une fonction récursive qui calcule la puissance n^{e} d'un nombre ;
2. Utiliser le fait que $x^{2n} = (x^2)^n$ pour améliorer la fonction.
3. Quelles sont les complexités temporelles des deux versions ?

Exercice

1. Écrire une fonction récursive qui calcule le terme d'indice n de la suite de Fibonacci ;
2. Quelle est la complexité temporelle de cette version récursive ?

Récursivité : Exercices

Exercice

La **hauteur palindromique** (en base 10) $h(n)$ d'un nombre entier n est :

- $h(n) = 0$ si n est un palindrome (p. ex. 454) ;
- $h(n) = 1$ si n n'est pas un palindrome mais que $n_1 = n + r(n)$ en est un, avec $r(n)$ l'image miroir de n (p. ex. r(123) = 321) ;
- $h(n) = 2$ si n et $n_1 = n + r(n)$ ne sont pas des palindromes mais que $n_2 = n_1 + r(n_1)$ en est un ;
- etc.

Questions :

1. En supposant donnée la fonction `r::Integer->Integer` écrire une fonction `hpal` qui calcule la hauteur palindromique d'un entier n en base 10 (lorsqu'elle est finie) ;
2. Écrire la fonction `r`.

Déclarations locales

- La plupart des langages fonctionnels autorisent des déclarations **locales** :

```
-- let ... in ... est une expression
sommeCube x y = let z = x+y in z*z*z
```

Déclarations locales

- La plupart des langages fonctionnels autorisent des déclarations **locales** :

```
    -- let ... in ... est une expression
    sommeCube x y = let z = x+y in z*z*z
```

- Y compris de fonctions :

```
    sommeCube x y = let add u v = u+v
                        z = add x y
                    in z*z*z
```

Déclarations locales

- La plupart des langages fonctionnels autorisent des déclarations **locales** :

```
-- let ... in ... est une expression
sommeCube x y = let z = x+y in z*z*z
```

- Y compris de fonctions :

```
sommeCube x y = let add u v = u+v
                    z = add x y
                in z*z*z
```

- Alternative (Haskell) :

```
signe' x | x > 0     = unit
         | x < 0     = -unit
         | otherwise = 0
         where -- porte sur toutes les conditions
           unit = 1
```

Plan

Types de base

- On se donne les types de base :

Type	Haskell
Entiers	`Integer` (non bornés) ou `Int` (32/64-bits)
Réels virgule flottante	`Double` ou `Float`
Booléens	`Bool` (valeurs `True` et `False`)
Caractères	`Char`
Chaîne de caractères	`String` (alias pour `[Char]`)

Types de base

- On se donne les types de base :

Type	Haskell
Entiers	`Integer` (non bornés) ou `Int` (32/64-bits)
Réels virgule flottante	`Double` ou `Float`
Booléens	`Bool` (valeurs `True` et `False`)
Caractères	`Char`
Chaîne de caractères	`String` (alias pour `[Char]`)

- Le **type d'une fonction** est donné par les types de son domaine et son codomaine :
  Le cas des fonctions de plusieurs variables sera expliqué plus loin

```
positif::Integer->Bool
positif n = (n > 0)
```

Types de base

- On se donne les types de base :

Type	Haskell
Entiers	`Integer` (non bornés) ou `Int` (32/64-bits)
Réels virgule flottante	`Double` ou `Float`
Booléens	`Bool` (valeurs `True` et `False`)
Caractères	`Char`
Chaîne de caractères	`String` (alias pour `[Char]`)

- Le **type d'une fonction** est donné par les types de son domaine et son codomaine :
  Le cas des fonctions de plusieurs variables sera expliqué plus loin

```
positif::Integer->Bool
positif n = (n > 0)
```

- Haskell possède des algorithmes assez poussés d'**inférence de type**.

Types algébriques

- À partir des types de base, on peut construire des types plus **complexes** ;

Types algébriques

- À partir des types de base, on peut construire des types plus **complexes** ;
- Types **sommes** : énumerations

```
data Bool = True | False
data Animal = Canard | Vache | Chat
```

Types algébriques

- À partir des types de base, on peut construire des types plus **complexes** ;
- Types **sommes** : énumerations

```
    data Bool = True | False
    data Animal = Canard | Vache | Chat
```

- Types **produits** : tuples

```
    data AnimalMesure = Paire Animal Integer
    data Entiers3 = Entiers3 Integer Integer Integer
```

Types algébriques

- À partir des types de base, on peut construire des types plus **complexes** ;
- Types **sommes** : énumerations

```
data Bool = True | False
data Animal = Canard | Vache | Chat
```

- Types **produits** : tuples

```
data AnimalMesure = Paire Animal Integer
data Entiers3 = Entiers3 Integer Integer Integer
```

- `Paire`, `True`, `False`, `Canard`, etc. sont des **constructeurs de données** ;

Types algébriques

- À partir des types de base, on peut construire des types plus **complexes** ;
- Types **sommes** : énumerations

```
data Bool = True | False
data Animal = Canard | Vache | Chat
```

- Types **produits** : tuples

```
data AnimalMesure = Paire Animal Integer
data Entiers3 = Entiers3 Integer Integer Integer
```

- `Paire`, `True`, `False`, `Canard`, etc. sont des **constructeurs de données** ;
- Ils peuvent avoir n'importe quel nom, y compris le même que le nom du type défini Comme pour `Entiers3` ;

Types algébriques

- À partir des types de base, on peut construire des types plus **complexes** ;
- Types **sommes** : énumerations

```
data Bool = True | False
data Animal = Canard | Vache | Chat
```

- Types **produits** : tuples

```
data AnimalMesure = Paire Animal Integer
data Entiers3 = Entiers3 Integer Integer Integer
```

- `Paire`, `True`, `False`, `Canard`, etc. sont des **constructeurs de données** ;
- Ils peuvent avoir n'importe quel nom, y compris le même que le nom du type défini Comme pour `Entiers3` ;
- Les noms de types et de constructeurs commencent par une **majuscule** en Haskell.

Types algébriques

- Types **algébriques** : sommes de produits

```
data PossibleReel = Rien | Valeur Double

inversion::Double->PossibleReel
inversion 0 = Rien
inversion x = Valeur (1/x)
```

Types algébriques

- Types **algébriques** : sommes de produits

```
data PossibleReel = Rien | Valeur Double

inversion::Double->PossibleReel
inversion 0 = Rien
inversion x = Valeur (1/x)
```

- Les constructeurs de données sont des fonctions :

```
-- Rien et Valeur sont *implicitement* définies avec ce type:
Rien::PossibleReel
Valeur::Double->PossibleReel
```

C'est la déclaration du type avec `data` qui les définit

Plus de *pattern-matching*

- On peut utiliser les **constructeurs** pour faire du **pattern-matching** :

```
opposePR::PossibleReel->PossibleReel
opposePR Rien       = Rien
opposePR (Valeur x) = Valeur (-x)

-- ou bien avec let ... in
opposePR z = if z == Rien
             then Rien
             else let (Valeur x)=z in Valeur (-x)
```

Plus de *pattern-matching*

- On peut utiliser les **constructeurs** pour faire du **pattern-matching** :

```
opposePR::PossibleReel->PossibleReel
opposePR Rien        = Rien
opposePR (Valeur x) = Valeur (-x)

-- ou bien avec let ... in
opposePR z = if z == Rien
             then Rien
             else let (Valeur x)=z in Valeur (-x)
```

Exercice

Écrire une fonction `et` qui réalise le *et logique* entre deux `Bool`

Types algébriques : Exercice

Exercice

1. Proposer la définition d'un type `Point` représentant un point du plan ;
2. Écrire une fonction `distance` qui donne la distance entre deux `Point` ;
3. Proposer la définition d'un type `Figure` qui est :
   - soit un unique `Point` ;
   - soit un cercle défini par son rayon > 0 et son centre ;
   - soit un carré défini par deux sommets opposés.
4. Écrire une fonction `perimetre` qui donne le périmètre d'une `Figure`.

Types paramétrés

- Le type `PossibleReel` existe, sous la forme d'un **type paramétré**, en Haskell :

```
-- pour tout type a...
data Maybe a = Nothing | Just a
```

Types paramétrés

- Le type `PossibleReel` existe, sous la forme d'un **type paramétré**, en Haskell :

```
-- pour tout type a...
data Maybe a = Nothing | Just a
```

- `Maybe` est un **constructeur de type** ;

Types paramétrés

- Le type `PossibleReel` existe, sous la forme d'un **type paramétré**, en Haskell :

```
-- pour tout type a...
data Maybe a = Nothing | Just a
```

- `Maybe` est un **constructeur de type** ;
- Il peut y avoir plusieurs paramètres :

```
-- pour tous types a et b...
data Either a b = Left a | Right b

safeTwice::Int->Either Int Int
safeTwice n = if n > (maxBound::Int) `div` 2
                then Left n
                else Right (2*n)
```

Types en Haskell : `type` et `newtype`

- `type` définit un type **synonyme** d'un autre :

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

- `newtype` est similaire à `data` mais ne fonctionne que pour un seul constructeur de données avec au plus une variable et fournit dans ce cas une petite optimisation (Synonymes mais crée un type distinct).

Types en Haskell : les tuples

- Un **tuple** est un type produit ;

Types en Haskell : les tuples

- Un **tuple** est un type produit ;
- Haskell propose une syntaxe spécifique (et classique) pour les tuples ;

```haskell
-- fst et snd pour les couples
maxi::(Integer,Integer)->Integer
maxi x = let a = fst x, b = snd x in
                 if a > b then a else b

-- pattern matching
maxi'::(Integer,Integer)->Integer
maxi' (a,b) = if a > b then a else b

-- pas que pour les couples
ror::(Char,Char,Char)->(Char,Char,Char)
ror (a,b,c) = (c,a,b)
```

Types en Haskell : les tuples

- Un **tuple** est un type produit ;
- Haskell propose une syntaxe spécifique (et classique) pour les tuples ;

```haskell
-- fst et snd pour les couples
maxi::(Integer,Integer)->Integer
maxi x = let a = fst x, b = snd x in
                 if a > b then a else b

-- pattern matching
maxi'::(Integer,Integer)->Integer
maxi' (a,b) = if a > b then a else b

-- pas que pour les couples
ror::(Char,Char,Char)->(Char,Char,Char)
ror (a,b,c) = (c,a,b)
```

- Les types tuples n'ont pas besoin de déclaration spécifique.

Types en Haskell : les enregistrements

- Un **enregistrement** est un type produit avec **accesseurs** intégrés (en lecture bien sûr) :

```
data Canard = Coin { nom::String, enverg::Double }

info::Canard->String
info c = "Oh le beau canard " ++ (nom c) ++
         " d'envergure " ++ show (enverg c) ++ "m "

main = let c1 = Coin "Coincoin" 0.8
           c2 = Coin { nom="Grocoin", enverg=0.8 }
           c3 = c2 { enverg = 1 }
       in
           print (info c1 ++ info c3)
```

Types algébriques récursifs

- Les types algébriques peuvent être **récursifs** :

```
-- arithmétique de Peano
data Nat = Zero | Succ Nat
```

Types algébriques récursifs

- Les types algébriques peuvent être **récursifs** :

```
-- arithmétique de Peano
data Nat = Zero | Succ Nat
```

Exercice

Écrire une fonction `addition` qui réalise l'addition de deux `Nat`.

Listes et arbres

Exercice

1. Écrire un type `Liste` représentant une liste d'entiers ;
2. Écrire une fonction `sum` qui fait la somme des éléments d'une liste d'entiers ;
3. Écrire un type `ArbreBinaire` représentant un arbre binaire d'entiers ;
4. Écrire une fonction `hauteur` qui calcule la hauteur d'un arbre binaire d'entiers.

Types en Haskell : les listes

- Le type **liste** est natif à Haskell :

Types en Haskell : les listes

- Le type **liste** est natif à Haskell :
- La liste vide est `[]` ;

Types en Haskell : les listes

- Le type **liste** est natif à Haskell :
- La liste vide est `[]` ;
- Le constructeur « élément plus liste donne liste » est noté `:` ;

Types en Haskell : les listes

- Le type **liste** est natif à Haskell :
- La liste vide est `[]` ;
- Le constructeur « élément plus liste donne liste » est noté `:` ;
- Pour tout type `a` le type « liste de `a` » est noté `[a]` ;

```
sum::[Integer]->Integer
sum [] = 0
sum (x:xs) = x + sum xs
```

Types en Haskell : les listes

- Le type **liste** est natif à Haskell :
- La liste vide est `[]` ;
- Le constructeur « élément plus liste donne liste » est noté `:` ;
- Pour tout type `a` le type « liste de `a` » est noté `[a]` ;

```
sum::[Integer]->Integer
sum [] = 0
sum (x:xs) = x + sum xs
```

- Accès à l'élément `i` (commençant à 0) : `xs!!i`.

Listes en compréhension

- Les listes en compréhension (*list comprehension*) sont une **facilité syntaxique** pour créer des listes :

```
-- les nombres pairs inférieurs à n (on peut faire mieux...)
evens n = [x+1 | x<-[1..(n-2)], odd x]

-- les produits de nombres impairs à trois chiffres
products3 = [m*n | m <- [101,103..999],
                   n <- [101,103..999]]
```

Listes en compréhension

- Les listes en compréhension (*list comprehension*) sont une **facilité syntaxique** pour créer des listes :

```
-- les nombres pairs inférieurs à n (on peut faire mieux...)
evens n = [x+1 | x<-[1..(n-2)], odd x]

-- les produits de nombres impairs à trois chiffres
products3 = [m*n | m <- [101,103..999],
                   n <- [101,103..999]]
```

Exercice

Construire la liste des triplets pythagoriciens (a, b, c), c.-à-d. tels que $a^2 + b^2 = c^2$, dont la somme des composantes vaut 1000 et tels que $a < b < c$. (En fait il n'y en a qu'un)

Exercices sur les listes

Exercice

Écrire une fonction `reverse` qui inverse une liste ;

Exercices sur les listes

Exercice

Écrire une fonction `reverse` qui inverse une liste ;

Exercice

1. Écrire une fonction `delete` qui efface la première occurrence d'un élément dans une liste ;
2. Écrire une fonction `maximum` qui trouve le maximum dans une liste d'entiers ;
3. Écrire une fonction `trimax` qui réalise le tri par extraction du maximum dans une liste d'entiers.

Exercices sur les listes

Exercice

Écrire une fonction `reverse` qui inverse une liste ;

Exercice

1. Écrire une fonction `delete` qui efface la première occurrence d'un élément dans une liste ;
2. Écrire une fonction `maximum` qui trouve le maximum dans une liste d'entiers ;
3. Écrire une fonction `trimax` qui réalise le tri par extraction du maximum dans une liste d'entiers.

`reverse` et `maximum` existent déjà dans le `Prelude` d'Haskell, `delete` et `sort` (par un autre algorithme) sont dans `Data.List`

Types génériques

- On peut définir les fonctions de façon **générique** :

```
-- pour tout type a
fsquare::(a->a)->(a->a)
fsquare f = f.f
```

Types génériques

- On peut définir les fonctions de façon **générique** :

```
-- pour tout type a
fsquare::(a->a)->(a->a)
fsquare f = f.f
```

- Et des **types** algébriques de façon générique (paramétrée) : pour tout type `a`, le type `[a]` est une liste de `a` et `Maybe a` est la possibilité d'un `a` ;

Types génériques

- On peut définir les fonctions de façon **générique** :

```
-- pour tout type a
fsquare::(a->a)->(a->a)
fsquare f = f.f
```

- Et des **types** algébriques de façon générique (paramétrée) : pour tout type `a`, le type `[a]` est une liste de `a` et `Maybe a` est la possibilité d'un `a` ;
- Et des fonctions génériques sur des types algébriques génériques :

```
-- pour tout type a
head::[a]->a
head [] = error ("head: Empty list") -- fct. partielle!
head (x:xs) = x

-- dans Data.Maybe
listToMaybe::[a]->Maybe a
listToMaybe [] = Nothing
listToMaybe (x:xs) = Just x
```

Plan

Fonctions d'ordre supérieur

- Les fonctions sont des objets de **première classe** : elles se manipulent comme les types de base ;

Fonctions d'ordre supérieur

- Les fonctions sont des objets de **première classe** : elles se manipulent comme les types de base ;
- Notion de fonction d'**ordre supérieur** : **ensembles de fonctions** dans le domaine ou le codomaine ;

```
-- applique f à tous les éléments
map _ [] = []
map f (x:xs) = (f x):(map f xs)
```

« Curryfication » (*Currying*)

- `max` est une fonction de deux entiers qui donne un entier :

```
    max x y = if x>y then x else y
```

« Curryfication » (*Currying*)

- `max` est une fonction de deux entiers qui donne un entier :

```
    max x y = if x>y then x else y
```

- Comment écrire son **type** ?

« Curryfication » (*Currying*)

- `max` est une fonction de deux entiers qui donne un entier :

```
max x y = if x>y then x else y
```

- Comment écrire son **type** ?
- `max::(Integer,Integer)->Integer` semble le plus propre. Mais on aurait :

```
max (x,y) = if x>y then x else y
```

« Curryfication » (*Currying*)

- `max` est une fonction de deux entiers qui donne un entier :

```
max x y = if x>y then x else y
```

- Comment écrire son **type** ?
- `max::(Integer,Integer)->Integer` semble le plus propre. Mais on aurait :

```
max (x,y) = if x>y then x else y
```

- Pour tout `x`, `(max x)` est une fonction d'un entier qui donne un entier

« Curryfication » (*Currying*)

- `max` est une fonction de deux entiers qui donne un entier :

```
    max x y = if x>y then x else y
```

- Comment écrire son **type** ?
- `max::(Integer,Integer)->Integer` semble le plus propre. Mais on aurait :

```
    max (x,y) = if x>y then x else y
```

- Pour tout `x`, `(max x)` est une fonction d'un entier qui donne un entier
- Donc `max` est aussi une fonction d'un entier qui donne « une fonction d'un entier qui donne un entier »

```
    max::Integer->(Integer->Integer)
```

« Curryfication » (*Currying*)

- `max` est une fonction de deux entiers qui donne un entier :

```
max x y = if x>y then x else y
```

- Comment écrire son **type** ?
- `max::(Integer,Integer)->Integer` semble le plus propre. Mais on aurait :

```
max (x,y) = if x>y then x else y
```

- Pour tout `x`, `(max x)` est une fonction d'un entier qui donne un entier
- Donc `max` est aussi une fonction d'un entier qui donne « une fonction d'un entier qui donne un entier »

```
max::Integer->(Integer->Integer)
```

- Et comme `->` est associatif à droite :

```
max::Integer->Integer->Integer
```

« Curryfication » (*Currying*)

Toute fonction de **plusieurs** *variables peut être transformée en une fonction d'***une seule** *variable*

« Curryfication » (*Currying*)

Toute fonction de **plusieurs** *variables peut être transformée en une fonction d'***une seule** *variable*

- Pour tous types `a,b,c`, on peut définir deux fonctions :

```
curry::((a,b)->c)->(a->(b->c))
uncurry::(a->(b->c))->((a,b)->c)
-- et on pourrait définir des versions pour tous
-- les tuples de types possibles
```

« Curryfication » (*Currying*)

Toute fonction de **plusieurs** *variables peut être transformée en une fonction d'***une seule** *variable*

- Pour tous types a,b,c, on peut définir deux fonctions :

```
curry::((a,b)->c)->(a->(b->c))
uncurry::(a->(b->c))->((a,b)->c)
-- et on pourrait définir des versions pour tous
-- les tuples de types possibles
```

- Les fonctions sont « curryfiées » (*curried*) par défaut en Haskell

« Curryfication » (*Currying*)

Toute fonction de **plusieurs** *variables peut être transformée en une fonction d'***une seule** *variable*

- Pour tous types a,b,c, on peut définir deux fonctions :

```
curry::((a,b)->c)->(a->(b->c))
uncurry::(a->(b->c))->((a,b)->c)
-- et on pourrait définir des versions pour tous
-- les tuples de types possibles
```

- Les fonctions sont « curryfiées » (*curried*) par défaut en Haskell

Exercice

1. Quel est le type de `map` ?
2. Écrire une fonction `flip`, avec son type, qui à partir d'une fonction d'arité 2, donne la même fonction avec les variables inversées. Par exemple :

```
-- affiche 2 et non pas -2
main = print  (flip (-) 1 3)
```

Application de fonction

- L'**application de fonction** est une fonction ;

Application de fonction

- L'**application de fonction** est une fonction ;
- Elle est notée `($)` en Haskell :

```
    ($)::(a->b)->a->b
    ($) f x = f x
```

Application de fonction

- L'**application de fonction** est une fonction ;
- Elle est notée `($)` en Haskell :

```
($)::(a->b)->a->b
($) f x = f x
```

- Elle s'utilise comme un opérateur **infixe** sans les parenthèses
  Comme toutes les fonctions dont le nom est composé de symboles
  Dans l'autre sens : `div x y` peut s'écrire ``x `div` y``

Application de fonction

- L'**application de fonction** est une fonction ;
- Elle est notée `($)` en Haskell :

```
($)::(a->b)->a->b
($) f x = f x
```

- Elle s'utilise comme un opérateur **infixe** sans les parenthèses
  Comme toutes les fonctions dont le nom est composé de symboles
  Dans l'autre sens : `div x y` peut s'écrire ``x `div` y``
- Elle est de faible priorité et associative à droite :

```
main = print (square (inc 2))

-- ou un peu plus concis
main = print $ square $ inc 2
```

Application de fonction

- L'**application de fonction** est une fonction ;
- Elle est notée `($)` en Haskell :

```
($)::(a->b)->a->b
($) f x = f x
```

- Elle s'utilise comme un opérateur **infixe** sans les parenthèses
  Comme toutes les fonctions dont le nom est composé de symboles
  Dans l'autre sens : `div x y` peut s'écrire ``x `div` y``
- Elle est de faible priorité et associative à droite :

```
main = print (square (inc 2))

-- ou un peu plus concis
main = print $ square $ inc 2
```

- Elle sert pour définir certaines fonctions d'ordre supérieur :

```
-- valeurs en 0 des fonctions f, g et h
map ($ 0) [f,g,h]
```

Fonctions anonymes (Fonctions λ)

- On peut définir des **fonctions anonymes**, appelées fonctions λ : P. ex. $\lambda x.(x + 1)$ ou $\lambda x.(\lambda y.(x + y))$
- En Haskell :

```haskell
inc x = x+1
map inc [1,2,3]

-- ou simplement
map (\x -> x+1) [1,2,3]

-- plusieurs variables:
add = (\x y -> x+y)
```

Fermetures (*Closures*)

- On quantifie implicitement x et y quand on écrit :

```
    f x y = x + y    -- ∀x,∀y
```

Fermetures (*Closures*)

- On quantifie implicitement `x` et `y` quand on écrit :

```
f x y = x + y   -- ∀x,∀y
```

- La variable `y` est dite **libre** dans :

```
f' x = x + y   -- ∀x
```

Fermetures (*Closures*)

- On quantifie implicitement `x` et `y` quand on écrit :

```
f x y = x + y    -- ∀x, ∀y
```

- La variable `y` est dite **libre** dans :

```
f' x = x + y    -- ∀x
```

- On ne peut pas évaluer `f' x` sans une valeur pour `y`
  D'ailleurs, toute seule, ce n'est pas une déclaration correcte

Fermetures (*Closures*)

- On quantifie implicitement `x` et `y` quand on écrit :

```
f x y = x + y    -- ∀x,∀y
```

- La variable `y` est dite **libre** dans :

```
f' x = x + y    -- ∀x
```

- On ne peut pas évaluer `f' x` sans une valeur pour `y`
  D'ailleurs, toute seule, ce n'est pas une déclaration correcte
- Une **fermeture** est une fonction avec variables libres plus une valeur pour ces variables libres :

```
-- Pour tout y, g y est une fermeture!
g y = (\x -> x + y)
```

Fermetures (*Closures*)

- On quantifie implicitement `x` et `y` quand on écrit :

```
f x y = x + y    -- ∀x,∀y
```

- La variable `y` est dite **libre** dans :

```
f' x = x + y    -- ∀x
```

- On ne peut pas évaluer `f' x` sans une valeur pour `y`
  D'ailleurs, toute seule, ce n'est pas une déclaration correcte
- Une **fermeture** est une fonction avec variables libres plus une valeur pour ces variables libres :

```
-- Pour tout y, g y est une fermeture!
g y = (\x -> x + y)
```

- En particulier, via la **curryfication**, on crée des fermetures.

Récursion sur les listes : `map`

- Plutôt que de faire de la récursivité explicite, on utilise plutôt, autant que possible, des fonctions de **plus haut niveau** ;

Récursion sur les listes : `map`

- Plutôt que de faire de la récursivité explicite, on utilise plutôt, autant que possible, des fonctions de **plus haut niveau** ;
- On a déjà vu `map` qui permet d'**appliquer** une fonction **à chaque élément** d'une liste :

```
map::(a->b)->[a]->[b]
map _ [] = []
map f (x:xs) = (f x):(map f xs)
```

Récursion sur les listes : `map`

- Plutôt que de faire de la récursivité explicite, on utilise plutôt, autant que possible, des fonctions de **plus haut niveau** ;
- On a déjà vu `map` qui permet d'**appliquer** une fonction **à chaque élément** d'une liste :

```
map::(a->b)->[a]->[b]
map _ [] = []
map f (x:xs) = (f x):(map f xs)
```

Exercice

En utilisant `map`, écrire une fonction `an` qui donne la liste de toutes les anagrammes d'un mot.
On pourra utiliser aussi :

- `concat::[[a]]->[a]`
- `delete::Eq a =>a->[a]->[a]`

Récursion sur les listes : `filter`

- `filter` permet de **ne garder que** les éléments d'une liste qui satisfont un certain **prédicat** :

```
filter::(a->Bool)->[a]->[a]
filter _ [] = []
filter f (x:xs) = if f x then x:(filter f xs)
                         else filter f xs
```

Récursion sur les listes : `filter`

- `filter` permet de **ne garder que** les éléments d'une liste qui satisfont un certain **prédicat** :

```
filter::(a->Bool)->[a]->[a]
filter _ [] = []
filter f (x:xs) = if f x then x:(filter f xs)
                              else filter f xs
```

Exercice

Écrire une fonction `premiers` qui donne la liste des entiers premiers inférieurs à `n`.
On pourra ne pas chercher à optimiser et :

- écrire une fonction `diviseurs` donnant une liste de diviseurs de `n` ;
- utiliser la syntaxe [x..y] pour la liste des entiers compris entre `x` et `y` ;
- et `rem::Integer->Integer->Integer` donnant le reste la division entière ;
- et `null::[a]->Bool` indiquant si une liste est vide ;
- ou `length::[a]->Int` donnant la taille d'une liste.

Récursion sur les listes : `fold`s

- `foldl` et `foldr` permettent de **réduire** une liste à **une valeur** en appliquant une fonction d'arité 2 récursivement à tous les éléments :

Récursion sur les listes : `folds`

- `foldl` et `foldr` permettent de **réduire** une liste à **une valeur** en appliquant une fonction d'arité 2 récursivement à tous les éléments :
- `foldl` réduit de la gauche vers la droite :

```
foldl::(b->a->b)->b->[a]->b
foldl f x [] = x
foldl f x (y:ys) = foldl f (f x y) ys
```

Récursion sur les listes : `folds`

- `foldl` et `foldr` permettent de **réduire** une liste à **une valeur** en appliquant une fonction d'arité 2 récursivement à tous les éléments :
- `foldl` réduit de la gauche vers la droite :

```
foldl::(b->a->b)->b->[a]->b
foldl f x [] = x
foldl f x (y:ys) = foldl f (f x y) ys
```

- `foldr` réduit de la droite vers la gauche :

```
foldr::(a->b->b)->b->[a]->b
foldr f x [] = x
foldr f x (y:ys) = f y (foldr f x ys)
```

Récursion sur les listes : `folds`

- `foldl` et `foldr` permettent de **réduire** une liste à **une valeur** en appliquant une fonction d'arité 2 récursivement à tous les éléments :
- `foldl` réduit de la gauche vers la droite :

```
foldl::(b->a->b)->b->[a]->b
foldl f x [] = x
foldl f x (y:ys) = foldl f (f x y) ys
```

- `foldr` réduit de la droite vers la gauche :

```
foldr::(a->b->b)->b->[a]->b
foldr f x [] = x
foldr f x (y:ys) = f y (foldr f x ys)
```

- `foldl1,foldr1::(a->a->a)->[a]->a` sont des versions sans élément de départ pour les listes non vides ;

Récursion sur les listes : `foldl`

- `foldl` réduit de la gauche vers la droite :

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

- On a donc :

```
foldl f z [x1, x2, ..., xn]
                == f (...(f (f z x1) x2)...) xn

-- ou en notation infixe
foldl f z [x1, x2, ..., xn]
                == (...((z `f` x1) `f` x2)...) `f` xn
```

Récursion sur les listes : `foldr`

- `foldr` réduit de la droite vers la gauche :

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

- On a donc :

```
foldr f z [x1, x2, ..., xn]
            == f x1 (f x2 (...(f xn z)...))

-- ou en notation infixe
foldr f z [x1, x2, ..., xn]
            == x1 `f` (x2 `f` (...(xn `f` z)...))
```

Récursion sur les listes : `foldS`

Exercice

Écrire les fonctions suivantes (et tester à la main sur quelques exemples) :

- `sum::[Integer]->Integer` qui donne la somme des éléments d'une liste ;
- `maximum::[Integer]->Integer` qui donne le maximum des éléments d'une liste ;
- `and::[Bool]->Bool` qui indique si tous les éléments sont vrais ;
- `any::(a->Bool)->[a]->Bool` qui indique si au moins un élément de la liste satisfait un prédicat donné ;
- `concat::[[a]]->[a]` qui concatène une liste de listes.

Plan

Évaluation stricte (ou *eager*)

Les arguments d'une fonction sont **toujours complètement évalués** *avant que ladite fonction ne soit évaluée.*

Évaluation stricte (ou *eager*)

Les arguments d'une fonction sont **toujours complètement évalués** *avant que ladite fonction ne soit évaluée.*

- C'est le cas de **la plupart** des langages fonctionnels ou non.

Évaluation stricte (ou *eager*)

Les arguments d'une fonction sont **toujours complètement évalués** *avant que ladite fonction ne soit évaluée.*

- C'est le cas de **la plupart** des langages fonctionnels ou non.
- P. ex. : Java, C, Python, Ruby, OCaml, Scheme, etc.

Évaluation stricte (ou *eager*)

Les arguments d'une fonction sont **toujours complètement évalués** *avant que ladite fonction ne soit évaluée.*

- C'est le cas de **la plupart** des langages fonctionnels ou non.
- P. ex. : Java, C, Python, Ruby, OCaml, Scheme, etc.
- Facilite le raisonnement sur l'ordre d'évaluation, et l'évaluation de performance.

Évaluation non-stricte (ou *lazy*)

Les arguments d'une fonction ne sont évalués que **lorsqu'on en a besoin**.

Évaluation non-stricte (ou *lazy*)

Les arguments d'une fonction ne sont évalués que **lorsqu'on en a besoin**.

- C'est le cas de Haskell, Scala, et quelques rares autres peu connus ;

Évaluation non-stricte (ou *lazy*)

Les arguments d'une fonction ne sont évalués que **lorsqu'on en a besoin***.*

- C'est le cas de Haskell, Scala, et quelques rares autres peu connus ;
- Un certain nombre de langages proposent des constructions spécifiques la permettant sur demande (p. ex. : C#, Ocaml, Scheme) ;

Évaluation non-stricte (ou *lazy*)

Les arguments d'une fonction ne sont évalués que **lorsqu'on en a besoin**.

- C'est le cas de Haskell, Scala, et quelques rares autres peu connus ;
- Un certain nombre de langages proposent des constructions spécifiques la permettant sur demande (p. ex. : C#, Ocaml, Scheme) ;
- Permet de raisonner sur des constructions **infinies** :

```
fibs::[Integer]
fibs = 0:1: zipWith (+) fibs (tail fibs)

-- le 1234567e nombre de Fibonacci
main = print $ fibs !! 1234567
```

Évaluation non-stricte (ou *lazy*)

Les arguments d'une fonction ne sont évalués que **lorsqu'on en a besoin***.*

- C'est le cas de Haskell, Scala, et quelques rares autres peu connus ;
- Un certain nombre de langages proposent des constructions spécifiques la permettant sur demande (p. ex. : C#, Ocaml, Scheme) ;
- Permet de raisonner sur des constructions **infinies** :

```haskell
fibs::[Integer]
fibs = 0:1: zipWith (+) fibs (tail fibs)

-- le 1234567e nombre de Fibonacci
main = print $ fibs !! 1234567
```

- Donne un **gain de performance** dans certains cas.

Évaluation non-stricte (ou *lazy*)

Exercice

Donner une construction de la liste (infinie) des nombres premiers, en ne testant que les diviseurs potentiels premiers inférieurs ou égaux à $\sqrt{n}$ pour tout n candidat.

On pourra :

- repartir des fonctions `premiers`, `diviseurs` et `square` ;
- utiliser `takeWhile::(a->Bool)->[a]->[a]` ;
- utiliser la syntaxe `[2..]` pour la liste infinie des entiers supérieurs ou égaux à 2 ;

Notons également au passage l'existence en Haskell de `take::Int->[a]->[a]`, ainsi que `drop` et `dropWhile`.

Évaluation non-stricte et `foldS`

- `foldr` peut fonctionner sur des listes infinies !

```
any::(a->Bool)->[a]->Bool
any f = (foldr (||) False).(map f)

main = do
    -- Ceci donne vrai
    print $ any even [1..]

    -- Mais ceci boucle indéfiniment
    print $ any (\x->False) [1..]
```

Évaluation non-stricte et `fold`s

- `foldr` peut fonctionner sur des listes infinies !

```
any::(a->Bool)->[a]->Bool
any f = (foldr (||) False).(map f)

main = do
    -- Ceci donne vrai
    print $ any even [1..]

    -- Mais ceci boucle indéfiniment
    print $ any (\x->False) [1..]
```

- Mais `foldr` n'est pas récursif terminal et peut donc remplir la mémoire avec les opérations en attente sur de grosses listes ;

Évaluation non-stricte et `folds`

- `foldr` peut fonctionner sur des listes infinies !

```
any::(a->Bool)->[a]->Bool
any f = (foldr (||) False).(map f)

main = do
    -- Ceci donne vrai
    print $ any even [1..]

    -- Mais ceci boucle indéfiniment
    print $ any (\x->False) [1..]
```

- Mais `foldr` n'est pas récursif terminal et peut donc remplir la mémoire avec les opérations en attente sur de grosses listes ;
- `foldl` ne peut pas fonctionner sur des listes infinies mais est **récursif terminal**.

Évaluation non-stricte et `fold`s

- `foldr` peut fonctionner sur des listes infinies !

```
any::(a->Bool)->[a]->Bool
any f = (foldr (||) False).(map f)

main = do
    -- Ceci donne vrai
    print $ any even [1..]

    -- Mais ceci boucle indéfiniment
    print $ any (\x->False) [1..]
```

- Mais `foldr` n'est pas récursif terminal et peut donc remplir la mémoire avec les opérations en attente sur de grosses listes ;
- `foldl` ne peut pas fonctionner sur des listes infinies mais est **récursif terminal**.
- Mais l'évaluation non stricte implique que l'évaluation de (f z x) est différée à chaque fois dans :

```
foldl f z (x:xs) = foldl f (f z x) xs
```

`foldl'` et `seq`

- On peut forcer l'évaluation de `f` avec `foldl'` :

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

- `seq::a->b->b` évalue son premier argument puis donne le deuxième comme résultat.

Plan

Entrées, Sorties en Haskell

- En Haskell, les E/S sont **explicites** dans le type, via le type paramétré `IO` :

```
-- Écrire à l'écran
-- () est le type dit unitaire qui représente "rien"
putStr::String->IO ()
putStrLn::String->IO () -- saute une ligne à la fin

-- Affichage "intelligent" pour les instance de Show
print::(Show a) => a -> IO ()
print = putStrLn.show

-- Lire au clavier
getLine::IO String -- une ligne
getChar::IO Char   -- un caractère

-- Dual de Show
read::(Read a)=>String->a
```

Entrées, Sorties en Haskell

- `IO` est une **monade** (cf. généricité avancée) ;
- L'utilisation de `IO` est **contaminante** : il n'existe pas de fonction de type `IO a -> a` ;
- On peut par contre appliquer une fonction de type `a -> IO b` à une valeur de type `IO a` avec l'opérateur **bind** :

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

```
-- écho
main::IO () -- main fait toujours des E/S et produit le
            -- type « unitaire » (c.-à-d. rien)
main = getLine >>= putStrLn
```

Entrées, Sorties en Haskell

- `IO` est une **monade** (cf. généricité avancée) ;
- L'utilisation de `IO` est **contaminante** : il n'existe pas de fonction de type `IO a -> a` ;
- On peut par contre appliquer une fonction de type `a -> IO b` à une valeur de type `IO a` avec l'opérateur **bind** :

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

```
-- écho
main::IO ()  -- main fait toujours des E/S et produit le
             -- type « unitaire » (c.-à-d. rien)
main = getLine >>= putStrLn
```

Exercice

Modifier le code précédent pour :

1. afficher l'écho suivi de la même ligne à l'envers : entrer `coin` affiche `coin nioc`.
2. lire deux lignes et les afficher dans l'ordre inverse : entrer `coin` puis `meuh` affiche `meuh coin`.

Entrées, Sorties en Haskell

- L'enchaînement d'opérations par >>= produit du code peu lisible ;
- Haskell propose la notation do pour simplifier :

```
echo2 :: IO ()
echo2 = getLine >>= (\x -> getLine
                >>= (\y -> putStrLn (y ++ "␣" ++ x)))

-- ou équivalent :
echo2' :: IO ()
echo2' = do
    x <- getLine
    y <- getLine
    putStrLn (y ++ "␣" ++ x)
```

- La dernière instruction doit avoir le type promis (ici `IO ()`), qui est nécessairement de la forme `IO a`.

Entrées, Sorties en Haskell

- La fonction `return :: a -> IO a` permet de promouvoir une valeur de type `a` dans la monade `IO` :

```
putStrLn' :: IO String -> IO ()
putStrLn' s = s >>= putStrLn

main = do
    putStrLn' getLine
    putStrLn' $ return "coin"
```

Entrées, Sorties en Haskell

- La fonction `return :: a -> IO a` permet de promouvoir une valeur de type `a` dans la monade `IO` :

```
putStrLn' :: IO String -> IO ()
putStrLn' s = s >>= putStrLn

main = do
    putStrLn' getLine
    putStrLn' $ return "coin"
```

Exercice

Écrire une fonction `ioLength` qui donne la longueur d'une chaîne de caractères lue au clavier.

Entrées, Sorties en Haskell

- La fonction `return :: a -> IO a` permet de promouvoir une valeur de type `a` dans la monade `IO` :

```
putStrLn' :: IO String -> IO ()
putStrLn' s = s >>= putStrLn

main = do
    putStrLn' getLine
    putStrLn' $ return "coin"
```

Exercice

Écrire une fonction `ioLength` qui donne la longueur d'une chaîne de caractères lue au clavier.

- Autant que possible, on essaie d'écrire des fonctions sans `IO` et de circonscrire les E/S dans un petit nombre de fonctions bien identifiées.

Entrées, Sorties en Haskell

Exercice

Le *chiffre de César* permet de chiffrer un message sur l'alphabet $\{a, \ldots, z\}$ en décalant chaque lettre de 13 rangs vers z (avec une rotation si besoin) : par exemple a donne n et s donne f.

1. Écrire une fonction `cesar` qui pour un entier représentant le décalage (classiquement 13 mais on généralise) et une chaîne de caractère produit la chaîne de caractères chiffrée comme ci-dessus. Par exemple, `cesar 13 "coin"` vaut `"pbva"` et `cesar 13 "pbva"` vaut `"coin"`.
   La fonction `ord :: Char -> Int` donne le rang d'une lettre (attention le rang de `'a'` n'est pas 0) et `chr :: Int -> Char` donnant la lettre correspondant à un rang. Ces fonctions sont disponibles dans `Data.Char`.
2. Écrire la fonction `main` permettant l'acquisition de l'entier et de la chaîne de caractères, appelant la fonction `cesar` et affichant le résultat.
3. Modifier la fonction `main` pour forcer l'utilisateur à entrer un entier compris entre 1 et 25. Utiliser `let x = ...` (sans `in`) pour définir une nouvelle valeur `x` dans un bloc `do`

Entrées, Sorties en Haskell

- Évaluation **non-stricte** des E/S :

```
-- Rien n'est affiché ici:
ess = [print "Coin", print "Meuh", print "Miaou"]

main = do
        ess!!2          -- Affiche Miaou
        sequence_ ess   -- Exécute tous les affichages
                        -- de la liste
```

Entrées, Sorties en Haskell

- Évaluation **non-stricte** des E/S :

```
-- Rien n'est affiché ici:
ess = [print "Coin", print "Meuh", print "Miaou"]

main = do
        ess!!2              -- Affiche Miaou
        sequence_ ess       -- Exécute tous les affichages
                            -- de la liste
```

- À propos de `sequence` :

```
sequence::(Monad m)=>[m a]->m [a]
sequence_::(Monad m)=>[m a]->m ()

-- On peut aussi utiliser sequence pour Maybe:
-- (En fait pour toute monade)
xs = sequence [Just 1, Just 3, Just 5]  -- Just [1,3,5]
ys = sequence [Just 1, Nothing, Just 5] -- Nothing
```

Plan

Classes de types (*Typeclasses*)

- Pour certaines fonctions, il faut imposer des contraintes sur les types génériques :

```
-- pour tout type numérique a
product::Num a=>[a]->a
product = foldl (*) 1
```

Classes de types (*Typeclasses*)

- Pour certaines fonctions, il faut imposer des contraintes sur les types génériques :

```
-- pour tout type numérique a
product::Num a=>[a]->a
product = foldl (*) 1
```

- La définition de `product` est bonne pour tout type `a` pour lequel la fonction `(*)::a->a->a` est définie ;

Classes de types (*Typeclasses*)

- Pour certaines fonctions, il faut imposer des contraintes sur les types génériques :

```
-- pour tout type numérique a
product::Num a=>[a]->a
product = foldl (*) 1
```

- La définition de `product` est bonne pour tout type `a` pour lequel la fonction `(*)::a->a->a` est définie ;
- Une **classe de types** définit un ensemble de fonctions que toute **instance** doit définir.

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

    -- Minimal complete definition: (==) or (/=)
    x /= y      =  not (x == y)
    x == y      =  not (x /= y)
```

Classes de types (*Typeclasses*)

```haskell
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)     :: a -> a -> a
    negate            :: a -> a
    abs, signum       :: a -> a
    fromInteger       :: Integer -> a

        -- Minimal complete definition:
        -- All, except negate or (-)
    x - y             =  x + negate y
    negate x          =  0 - x
```

Instances

- On peut ensuite définir des **instances** de classes :

```
data Animal = Canard | Vache

instance Eq Animal where
    Canard == Canard    = True
    Canard == Vache     = False
    Vache  == Canard    = False
    Vache  == Vache     = True
```

Instances

- On peut ensuite définir des **instances** de classes :

```
data Animal = Canard | Vache

instance Eq Animal where
    Canard == Canard    = True
    Canard == Vache     = False
    Vache  == Canard    = False
    Vache  == Vache     = True
```

- Ou instances automatiques pour `Eq`, `Ord` (comparaisons) `Enum` ([x..y]), `Bounded` (`maxBound::a`, `minBound::a`), `Show` (`show::a->String`), ou `Read` (`read::String->a`) :

```
data Animal = Canard | Vache deriving (Eq, Ord, Show)
```

Foncteur (*functor*)

Exercice

1. Définir un type algébrique `Expr a` pour représenter une expression arithmétique sur des valeurs de type `a` et comportant les opérateurs d'incrément, de décrément, d'inversion, et de négation ;
2. Définir une fonction `evaluate` qui étant donnée une expression de type `Expr a` renvoit la valeur de type `a` correspondante ;
3. Tester avec l'expression correspondant à `dec(-(inc(4)))`. Et avec `1/0` ?
4. On veut éviter les valeurs infinies et les traiter explicitement à part :
   - définir une fonction `isZero` qui indique si un `Maybe a` vaut 0 ;
   - définir une fonction `mevaluate` qui renvoit cette fois un `Maybe a`, qui sera `Nothing` ssi l'expression contient une division par `0`.
5. Les traitements pour les 4 opérateurs sont similaires : écrire une fonction d'**ordre supérieur** `mfmap`, qui étant donnés un opérateur de type `a->b` (incrément, décrément, etc.), ainsi qu'une valeur de type `Maybe a` renvoit le résultat de type `Maybe b`.
6. Simplifier la fonction `mevaluate` en utilisant `mfmap`.

Foncteur (*functor*)

- `Maybe`, muni de `mfmap`, est appelé un **foncteur** ;

Foncteur (*functor*)

- `Maybe`, muni de `mfmap`, est appelé un **foncteur** ;
- Cette construction est **généralisable** pour d'autres constructeurs de type ;

Foncteur (*functor*)

- `Maybe`, muni de `mfmap`, est appelé un **foncteur** ;
- Cette construction est **généralisable** pour d'autres constructeurs de type ;
- Elle permet de **promouvoir** (*lift*) une fonction de type `a->b` en une fonction de type `m a->m b`.

Foncteur (*functor*)

- `Maybe`, muni de `mfmap`, est appelé un **foncteur** ;
- Cette construction est **généralisable** pour d'autres constructeurs de type ;
- Elle permet de **promouvoir** (*lift*) une fonction de type `a->b` en une fonction de type `m a->m b`.
- Haskell possède une classe de types `Functor`, dans laquelle la fonction de promotion s'appelle `fmap` ;

Foncteur (*functor*)

- `Maybe`, muni de `mfmap`, est appelé un **foncteur** ;
- Cette construction est **généralisable** pour d'autres constructeurs de type ;
- Elle permet de **promouvoir** (*lift*) une fonction de type `a->b` en une fonction de type `m a->m b`.
- Haskell possède une classe de types `Functor`, dans laquelle la fonction de promotion s'appelle `fmap` ;
- `fmap` doit vérifier les propriétés suivantes :

Foncteur (*functor*)

- `Maybe`, muni de `mfmap`, est appelé un **foncteur** ;
- Cette construction est **généralisable** pour d'autres constructeurs de type ;
- Elle permet de **promouvoir** (*lift*) une fonction de type `a->b` en une fonction de type `m a->m b`.
- Haskell possède une classe de types `Functor`, dans laquelle la fonction de promotion s'appelle `fmap` ;
- `fmap` doit vérifier les propriétés suivantes :
  1. `fmap id == id`

Foncteur (*functor*)

- `Maybe`, muni de `mfmap`, est appelé un **foncteur** ;
- Cette construction est **généralisable** pour d'autres constructeurs de type ;
- Elle permet de **promouvoir** (*lift*) une fonction de type `a->b` en une fonction de type `m a->m b`.
- Haskell possède une classe de types `Functor`, dans laquelle la fonction de promotion s'appelle `fmap` ;
- `fmap` doit vérifier les propriétés suivantes :
  1. `fmap id == id`
  2. `fmap (f.g) == (fmap f).(fmap g)`

Foncteur (*functor*)

- `Maybe`, muni de `mfmap`, est appelé un **foncteur** ;
- Cette construction est **généralisable** pour d'autres constructeurs de type ;
- Elle permet de **promouvoir** (*lift*) une fonction de type `a->b` en une fonction de type `m a->m b`.
- Haskell possède une classe de types `Functor`, dans laquelle la fonction de promotion s'appelle `fmap` ;
- `fmap` doit vérifier les propriétés suivantes :
  1. `fmap id == id`
  2. `fmap (f.g) == (fmap f).(fmap g)`
- `Maybe`, `[]`, `r->` ont des instances de la classe `Functor`.

Foncteur (*functor*)

- `Maybe`, muni de `mfmap`, est appelé un **foncteur** ;
- Cette construction est **généralisable** pour d'autres constructeurs de type ;
- Elle permet de **promouvoir** (*lift*) une fonction de type `a->b` en une fonction de type `m a->m b`.
- Haskell possède une classe de types `Functor`, dans laquelle la fonction de promotion s'appelle `fmap` ;
- `fmap` doit vérifier les propriétés suivantes :
  1. `fmap id == id`
  2. `fmap (f.g) == (fmap f).(fmap g)`
- `Maybe`, `[]`, `r->` ont des instances de la classe `Functor`.

Exercice

1. Vérifier que la fonction `mfmap` satisfait ces deux propriétés ;
2. Définir la fonction `fmap` pour le constructeur de type `[]` ;
3. Définir la fonction `fmap` pour le constructeur de type `r->`.

Foncteur applicatif (*applicative functor*)

Exercice

1. ajouter l'addition, la soustraction, la multiplication et la division au type `Expr a` ;
2. compléter `mevaluate`, en faisant apparaître comme précédemment une fonction `mliftA2` qui joue le rôle de `mfmap` pour les fonctions à deux paramètres.

Foncteur applicatif (*applicative functor*)

- Pour des fonctions à 3 paramètres, il faut une fonction `liftA3`, etc.

Foncteur applicatif (*applicative functor*)

- Pour des fonctions à 3 paramètres, il faut une fonction `liftA3`, etc.
- On peut profiter de la currification pour obtenir quelque chose de plus générique ;

Foncteur applicatif (*applicative functor*)

- Pour des fonctions à 3 paramètres, il faut une fonction `liftA3`, etc.
- On peut profiter de la currification pour obtenir quelque chose de plus générique ;
- Appliquons `fmap` à une fonction de plusieurs variables :

```
add :: Int->(Int->Int)           -- add est currifiée
fmap :: (a->b)->Maybe a->Maybe b -- pour mémoire

(fmap add) :: Maybe Int->Maybe (Int->Int) -- Argh!
-- on voudrait Maybe Int->Maybe Int->Maybe Int
```

Foncteur applicatif (*applicative functor*)

- Pour des fonctions à 3 paramètres, il faut une fonction `liftA3`, etc.
- On peut profiter de la currification pour obtenir quelque chose de plus générique ;
- Appliquons `fmap` à une fonction de plusieurs variables :

```
add :: Int->(Int->Int)          -- add est currifiée
fmap :: (a->b)->Maybe a->Maybe b -- pour mémoire

(fmap add) :: Maybe Int->Maybe (Int->Int) -- Argh!
-- on voudrait Maybe Int->Maybe Int->Maybe Int
```

- Il nous faut un opérateur supplémentaire :

```
apm :: Maybe (a -> b) -> Maybe a -> Maybe b

-- exemple:
fmap add :: Maybe Int -> Maybe (Int -> Int)
fmap add (Just 2) :: Maybe (Int -> Int)
apm (fmap add (Just 2)) :: Maybe Int -> Maybe Int
fmap add (Just 2) `apm` Nothing :: Maybe Int
```

Foncteur applicatif (*applicative functor*)

Exercice

1. définir `apm` ;
2. remplacer `mliftA2` par `mfmap` et `apm` ;
3. écrire une fonction `add3` qui somme trois éléments de type `a` ;
4. ajouter un opérateur d'addition à 3 éléments dans `Expr a` et compléter `mevaluate`.

Foncteur applicatif (*applicative functor*)

- Soit `mpure :: a -> Maybe a` la fonction telle que `mpure x = Just x` ;

Foncteur applicatif (*applicative functor*)

- Soit `mpure :: a -> Maybe a` la fonction telle que `mpure x = Just x` ;
- `mpure` transforme *a minima* un `a` en `Maybe a` et on peut l'utiliser à la place de `mfmap` avec `apm` :

```
-- les deux sont équivalents
mfmap (+) (Maybe 1) `apm` Nothing
mpure (+) `apm` (Maybe 1) `apm` Nothing
```

Foncteur applicatif (*applicative functor*)

- Soit `mpure :: a -> Maybe a` la fonction telle que `mpure x = Just x` ;
- `mpure` transforme *a minima* un `a` en `Maybe a` et on peut l'utiliser à la place de `mfmap` avec `apm` :

```
-- les deux sont équivalents
mfmap (+) (Maybe 1) `apm` Nothing
mpure (+) `apm` (Maybe 1) `apm` Nothing
```

- Alors `Maybe` muni de `mpure` et `apm` est un **foncteur applicatif** ;

Foncteur applicatif (*applicative functor*)

- Soit `mpure :: a -> Maybe a` la fonction telle que `mpure x = Just x` ;
- `mpure` transforme *a minima* un `a` en `Maybe a` et on peut l'utiliser à la place de `mfmap` avec `apm` :

```
-- les deux sont équivalents
mfmap (+) (Maybe 1) `apm` Nothing
mpure (+) `apm` (Maybe 1) `apm` Nothing
```

- Alors `Maybe` muni de `mpure` et `apm` est un **foncteur applicatif** ;
- Cette construction est généralisable et permet de promouvoir des fonctions à plus de un paramètres.

Foncteur applicatif (*applicative functor*)

- Soit `mpure :: a -> Maybe a` la fonction telle que `mpure x = Just x` ;
- `mpure` transforme *a minima* un `a` en `Maybe a` et on peut l'utiliser à la place de `mfmap` avec `apm` :

```
-- les deux sont équivalents
mfmap (+) (Maybe 1) `apm` Nothing
mpure (+) `apm` (Maybe 1) `apm` Nothing
```

- Alors `Maybe` muni de `mpure` et `apm` est un **foncteur applicatif** ;
- Cette construction est généralisable et permet de promouvoir des fonctions à plus de un paramètres.
- Haskell définit une classe de type `Applicative` dont les instances implémentent `pure` et `ap` (aussi notée `(<*>)`).

Foncteur applicatif (*applicative functor*)

- Soit `mpure :: a -> Maybe a` la fonction telle que `mpure x = Just x` ;
- `mpure` transforme *a minima* un `a` en `Maybe a` et on peut l'utiliser à la place de `mfmap` avec `apm` :

```
-- les deux sont équivalents
mfmap (+) (Maybe 1) `apm` Nothing
mpure (+) `apm` (Maybe 1) `apm` Nothing
```

- Alors `Maybe` muni de `mpure` et `apm` est un **foncteur applicatif** ;
- Cette construction est généralisable et permet de promouvoir des fonctions à plus de un paramètres.
- Haskell définit une classe de type `Applicative` dont les instances implémentent `pure` et `ap` (aussi notée `(<*>)`).
- Ces deux fonctions doivent satisfaire :

```
pure id <*> v = v                        -- identité
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
                                         -- composition
pure f <*> pure x = pure (f x)           -- homomorphisme
u <*> pure y = pure ($ y) <*> u          -- interchangeabilité
```

Foncteur applicatif (*applicative functor*)

- Tout foncteur applicatif est un foncteur :

Foncteur applicatif (*applicative functor*)

- Tout foncteur applicatif est un foncteur :

Exercice

Définir `fmap` en fonction de `pure` et `<*>`

Foncteur applicatif (*applicative functor*)

- Tout foncteur applicatif est un foncteur :

Exercice

Définir `fmap` en fonction de `pure` et `<*>`

- Pour appliquer une fonction « normale » à des valeurs « complexes » :

```
-- pour un paramètre
(fmap f) u
-- pour deux paramètres
fmap f u <*> v
f <$> u <*> v                -- (<$>) = fmap
pure f <*> u <*> v
(liftA2 f) u v               -- liftA2 f u v = f <$> u <*> v
-- pour 3 paramètres
f <$> u <*> v <*> w
pure f <*> u <*> v <*> w
(liftA3 f) u v w
...
```

Foncteur applicatif : le cas de `r->`

- On peut définir `pure` et `<*>` pour que `r->` soit un foncteur applicatif :

```
pure::a->(r->a)
pure x = (\_ -> x)

(<*>)::(r->(a->b))->(r->a)->(r->b)
f <*> u = (\x -> f x (u x))
```

Foncteur applicatif : le cas de `r->`

- On peut définir `pure` et `<*>` pour que `r->` soit un foncteur applicatif :

```
pure::a->(r->a)
pure x = (\_ -> x)

(<*>)::(r->(a->b))->(r->a)->(r->b)
f <*> u = (\x -> f x (u x))
```

Exercice

Donner une expression sans variable ni λ (*point-free*) des fonctions :

1. `f x = (cos x) * x`.
2. `g x = (cos x) * (sin x)`.

Monade (*Monad*)

- On s'intéresse aux fonctions de la forme `a->m b`
  Résultats incertains (`[]`), manquants (`Maybe`), utilisation des entrées - sorties (`IO`), etc.

Monade (*Monad*)

- On s'intéresse aux fonctions de la forme `a->m b`
  Résultats incertains (`[]`), manquants (`Maybe`), utilisation des entrées - sorties (`IO`), etc.
- Comment **composer** ces fonctions ?

```
f::a->Maybe b
g::b->Maybe c -- b ≠ Maybe b: comment définir g.f?
```

Monade (*Monad*)

- On s'intéresse aux fonctions de la forme `a->m b`
  Résultats incertains (`[]`), manquants (`Maybe`), utilisation des entrées - sorties (`IO`), etc.
- Comment **composer** ces fonctions ?

```
f::a->Maybe b
g::b->Maybe c -- b ≠ Maybe b: comment définir g.f?
```

- On définit une nouvelle fonction pour nous aider :

```
-- pour Maybe
(=<<)::(a->Maybe b)->Maybe a->Maybe b
f =<< Nothing  = Nothing
f =<< (Just x) = f x

-- pour []
(=<<)::(a->[b])->[a]->[b]
(=<<) = concatMap -- concatMap f == concat.(map f)
```

Monade (*Monad*)

- On peut alors définir un opérateur de composition :

```
(=<<)::(a->m b)->m a->m b -- pour mémoire

(<=<)::(b->m c)->(a->m b)->(a->m c)
f <=< g = (f =<<).g         -- c.-à-d. ((=«) f).g
```

Monade (*Monad*)

- On peut alors définir un opérateur de composition :

```
(=<<)::(a->m b)->m a->m b -- pour mémoire

(<=<)::(b->m c)->(a->m b)->(a->m c)
f <=< g = (f =<<).g          -- c.-à-d. ((=«) f).g
```

- En Haskell, on utilise plutôt la version mirroir >>= (prononcer *bind*) :

```
(>>=)::m a->(a->m b)->m b
(>>=) = flip (=<<)
```

Monade (*Monad*)

- On peut alors définir un opérateur de composition :

```
(=<<)::(a->m b)->m a->m b -- pour mémoire

(<=<)::(b->m c)->(a->m b)->(a->m c)
f <=< g = (f =<<).g         -- c.-à-d. ((=«) f).g
```

- En Haskell, on utilise plutôt la version mirroir >>= (prononcer *bind*) :

```
(>>=)::m a->(a->m b)->m b
(>>=) = flip (=<<)
```

- Exemple d'utilisation :

```
roll :: Integer->[Integer]
roll x = [x + y | y <- [1..6]]

roll 0 -- résultats du lancer d'un 1 dé à 6 faces
roll 0 >>= roll -- deux lancers (avec multiplicités)
roll 0 >>= roll >>= roll -- etc.
```

Monade (*Monad*)

- On définit aussi `pure::a->m a` mais on l'appelle parfois `return` :

```
return::a->m a
return = pure
```

Monade (*Monad*)

- On définit aussi `pure::a->m a` mais on l'appelle parfois `return` :

```
return::a->m a
return = pure
```

- `m` est une **monade** si :

```
return a >>= f            == f a
m >>= return              == m
m >>= (\x -> f x >>= g)   == (m >>= f) >>= g
```

Monade (*Monad*)

- On définit aussi `pure::a->m a` mais on l'appelle parfois `return` :

```
return::a->m a
return = pure
```

- `m` est une **monade** si :

```
return a >>= f            == f a
m >>= return              == m
m >>= (\x -> f x >>= g)   == (m >>= f) >>= g
```

- Si `m` est une monade, on parle de :

Monade (*Monad*)

- On définit aussi `pure::a->m a` mais on l'appelle parfois `return` :

```
return::a->m a
return = pure
```

- `m` est une **monade** si :

```
return a >>= f            == f a
m >>= return              == m
m >>= (\x -> f x >>= g)   == (m >>= f) >>= g
```

- Si `m` est une monade, on parle de :
  - **valeur monadique** pour une valeur de type `m a` ;

Monade (*Monad*)

- On définit aussi `pure::a->m a` mais on l'appelle parfois `return` :

```
return::a->m a
return = pure
```

- `m` est une **monade** si :

```
return a >>= f          == f a
m >>= return            == m
m >>= (\x -> f x >>= g)  == (m >>= f) >>= g
```

- Si `m` est une monade, on parle de :
  - **valeur monadique** pour une valeur de type `m a` ;
  - **fonction monadique** pour une fonction de type `a->m b` ;

Monade (*Monad*)

- On définit aussi `pure::a->m a` mais on l'appelle parfois `return` :

```
return::a->m a
return = pure
```

- `m` est une **monade** si :

```
return a >>= f            == f a
m >>= return              == m
m >>= (\x -> f x >>= g)   == (m >>= f) >>= g
```

- Si `m` est une monade, on parle de :
  - **valeur monadique** pour une valeur de type `m a` ;
  - **fonction monadique** pour une fonction de type `a->m b` ;
- Haskell propose la classe de type `Monad`.

Monades et foncteurs

- Toute monade est un foncteur

Monades et foncteurs

- Toute monade est un foncteur

Exercice

Définir `fmap` à partir de >>= et `return`.

Monades et foncteurs

- Toute monade est un foncteur

Exercice

Définir `fmap` à partir de >>= et `return`.

- Toute monade est un foncteur applicatif

```
f <*> u == f >>= (\g -> fmap g u)
f <*> u == f >>= (\g -> u >>= return.g)
f >>= \g -> u >>= \x -> return (g x)
```

Monade : combinaison de valeurs monadiques

- On peut définir (entre autres) `liftM2` = `liftA2` :

```
inverse 0 = Nothing
inverse x = Just $ 1 / x

-- Faire inverse x + inverse y
somme' x y = liftM2 (+) (inverse x) (inverse y)
```

Monade : combinaison de valeurs monadiques

- On peut définir (entre autres) `liftM2 = liftA2` :

```
inverse 0 = Nothing
inverse x = Just $ 1 / x

-- Faire inverse x + inverse y
somme' x y = liftM2 (+) (inverse x) (inverse y)
```

Exercice

Écrire une fonction `sommeInverses` qui fait la somme des inverses de deux réels (peut échouer), en n'utilisant que `>>=` et `return` (et donc ni `fmap`, ni `liftM2` ou autres).

Monade : combinaison de valeurs monadiques

- On peut définir (entre autres) `liftM2 = liftA2` :

```
inverse 0 = Nothing
inverse x = Just $ 1 / x

-- Faire inverse x + inverse y
somme' x y = liftM2 (+) (inverse x) (inverse y)
```

Exercice

Écrire une fonction `sommeInverses` qui fait la somme des inverses de deux réels (peut échouer), en n'utilisant que `>>=` et `return` (et donc ni `fmap`, ni `liftM2` ou autres).

- Solution la plus simple

```
sommeInverses::Double->Double->Maybe Double
sommeInverses x y = inverse x >>=
              (\x' -> inverse y >>=
               \y' -> return (x'+y'))
```

On peut y arriver directement ou à partir de `liftM2` en appliquant les lois des monades.

Monade : la notation `do` de Haskell

- On peut **enchaîner** les fonctions monadiques de la façon précédente :

```
f::a->m b
g::c->m d
h::b->d->m e
k::a->c->m e -- on veut k x y = h (f x) (g y)
k x y = f x >>= (\x' -> g y >>= (\y' -> h x' y'))
```

Monade : la notation do de Haskell

- On peut **enchaîner** les fonctions monadiques de la façon précédente :

```
f::a->m b
g::c->m d
h::b->d->m e
k::a->c->m e -- on veut k x y = h (f x) (g y)
k x y = f x >>= (\x' -> g y >>= (\y' -> h x' y'))
```

- Généralisable à un nombre de fonctions monadiques quelconque ;

Monade : la notation `do` de Haskell

- On peut **enchaîner** les fonctions monadiques de la façon précédente :

```
f::a->m b
g::c->m d
h::b->d->m e
k::a->c->m e -- on veut k x y = h (f x) (g y)
k x y = f x >>= (\x' -> g y >>= (\y' -> h x' y'))
```

- Généralisable à un nombre de fonctions monadiques quelconque ;
- Haskell fournit une **notation spécifique** pour cette construction :

```
k x y = do
            x' <- f x
            y' <- g y
            h x' y'
```

Monade : la notation `do` de Haskell

- On peut **enchaîner** les fonctions monadiques de la façon précédente :

```
f::a->m b
g::c->m d
h::b->d->m e
k::a->c->m e -- on veut k x y = h (f x) (g y)
k x y = f x >>= (\x' -> g y >>= (\y' -> h x' y'))
```

- Généralisable à un nombre de fonctions monadiques quelconque ;
- Haskell fournit une **notation spécifique** pour cette construction :

```
k x y = do
            x' <- f x
            y' <- g y
            h x' y'
```

- la dernière expression doit évidemment fournir le type promis.

Monade : la notation do de Haskell

- On peut **enchaîner** les fonctions monadiques de la façon précédente :

```
f::a->m b
g::c->m d
h::b->d->m e
k::a->c->m e -- on veut k x y = h (f x) (g y)
k x y = f x >>= (\x' -> g y >>= (\y' -> h x' y'))
```

- Généralisable à un nombre de fonctions monadiques quelconque ;
- Haskell fournit une **notation spécifique** pour cette construction :

```
k x y = do
            x' <- f x
            y' <- g y
            h x' y'
```

- la dernière expression doit évidemment fournir le type promis.

Exercice

Réécrire `sommeInverses` en utilisant la notation `do`.

Monade : la notation `do` de Haskell

- La construction précédente **séquentialise** l'évaluation des fonctions monadiques ;

Monade : la notation `do` de Haskell

- La construction précédente **séquentialise** l'évaluation des fonctions monadiques ;
- si la valeur d'une de ces fonctions n'a pas d'intérêt, on peut utiliser `>>` au lieu de `>>=` :

```
(>>)::m a->m b->m b
u >> v = u >>= \_ -> v
```

Monade : la notation do de Haskell

- La construction précédente **séquentialise** l'évaluation des fonctions monadiques ;
- si la valeur d'une de ces fonctions n'a pas d'intérêt, on peut utiliser >> au lieu de >>= :

```
(>>)::m a->m b->m b
u >> v = u >>= \_ -> v
```

- Exemple :

```
sommeInverses' x y = inverse x
                     >>= (\x' -> Just "Coin"
                     >> inverse y
                     >>= (\y' -> return (x'+y')))

-- ou de façon équivalente
sommeInverses' x y = do
                          x'<-inverse x
                          Just "Coin"
                          y'<-inverse y
                          return x'+y'
```

Plan

Conclusion

- La programmation fonctionnelle est un paradigme qui a plusieurs avantages :
  - Facilités de raisonnement sur la correction ;
  - Facilités de parallélisation ;
  - Concision et lisibilité, notamment grâce aux fonctions d'ordre supérieur.

Conclusion

- La programmation fonctionnelle est un paradigme qui a plusieurs avantages :
  - Facilités de raisonnement sur la correction ;
  - Facilités de parallélisation ;
  - Concision et lisibilité, notamment grâce aux fonctions d'ordre supérieur.
- Et quelques inconvénients :
  - Pas de notion d'état « mutable » $\Rightarrow$ certaines tâches sont plus difficiles à faire efficacement
    Les différents langages proposent des solutions à ce problème
  - Raisonnement assez différent de la programmation séquentielle.

Pour aller plus loin

- Sur la théorie :
  - λ-calcul ;
  - Catégories ;
  - Structures de données purement fonctionnelles ;

Pour aller plus loin

- Sur la théorie :
  - λ-calcul ;
  - Catégories ;
  - Structures de données purement fonctionnelles ;
- Sur Haskell :
  - Plus de monades (notamment **ST** pour la mutabilité) ;
  - Transformateurs de monades (pour utiliser plusieurs monades en même temps) ;
  - Évaluation non stricte – *Weak Head Normal Form*.