Chapitre 1 : Introduction

Carole Porrier
carole.porrier@univ-paris13.fr

Département d'informatique
IUT de Villetaneuse

14 mars 2024
Paradigmes de développement universels

Table des matières

1. Présentation du cours
2. Paradigmes de programmation
3. Rappels mathématiques
4. Le langage Haskell
5. Fonctions et types
6. Listes et tuples

Table des matières

1. Présentation du cours

2. Paradigmes de programmation

3. Rappels mathématiques

4. Le langage Haskell

5. Fonctions et types

6. Listes et tuples

Description du cours (1/2)

- Présenter les concepts fondamentaux de langages de **programmation modernes**.
- Comprendre les **possibilités** et **limites** des divers types de langages.
- Familiariser l'étudiant avec **différents paradigmes** de programmation et favoriser l'acquisition de **nouvelles techniques et stratégies** de programmation.
- Étude des paradigmes de programmation **fonctionnel** et **logique**.

Description du cours (2/2)

Revue des principes de **programmation fonctionnelle** :

- Stratégies d'**évaluation** des arguments.
- **Polymorphisme** et **déduction** des types.
- Fonctions d'**ordre supérieur**.
- **Efficacité** et optimisation.

Revue des principes de **programmation logique**.

- Forme **clausale** de la **logique du premier ordre** et clauses de Horn.
- **Unification** et **résolution**.
- Le problème de la **négation**.

Références

- Learn You a Haskell for Great Good!, par Miran Lipovača, disponible en ligne.
- Apprendre Haskell vous fera le plus grand bien !, traduction du livre de Miran Lipovača par Valentin Robert, disponible en ligne.
- Real World Haskell, par Bryan O'Sullivan, Don Stewart et John Goerzen, disponible en ligne.
- Purely Functional Data Structures, par Chris Osaki, disponible en ligne.
- Programming in Prolog, par William F. Clocksin et Christopher S. Mellish, Springer.
- Prolog: A Tutorial Introduction, par James Lu et Jerud J. Mead, disponible en ligne.

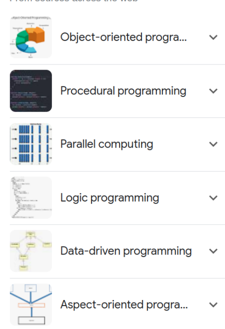
Table des matières

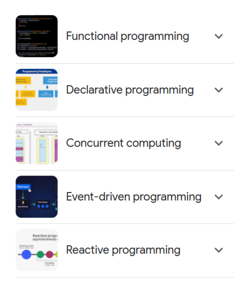
1. Présentation du cours
2. Paradigmes de programmation
3. Rappels mathématiques
4. Le langage Haskell
5. Fonctions et types
6. Listes et tuples

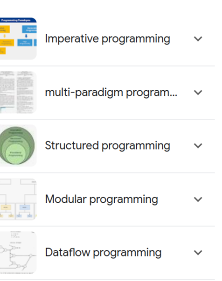
Plusieurs paradigmes

Programming Language Paradigms

From sources across the web







(source : Google)

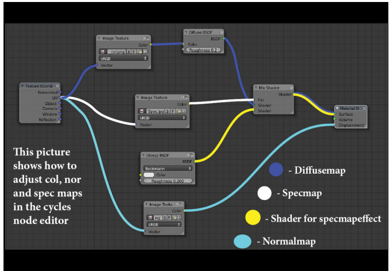
Paradigmes populaires

- **Impératif** (C, Pascal, Fortran, ...): les instructions sont exécutées **successivement** et modifient l'**état** courant du programme; inclut le **procédural**: définit des procédures ou des fonctions pour des tâches spécifiques;
- **Fonctionnel** (Haskell, Common Lisp, Scheme, ...): combinaison de **fonctions pures**, d'**ordre supérieur**, sans **effet de bord**;
- **Objet** (Java, C++, Smalltalk, etc.): des **entités** (les objets) interagissent entre elles, selon certaines règles;
- **Logique** (Prolog, ...): les problèmes sont modélisés par des **règles logiques**, desquels on peut faire de l'**inférence**.
- **Concurrent** (Erlang, Java, C#, ...): différents calculs sont fait en **parallèle**, dans un **ordre** quelconque.
- etc.

Paradigme “visuel”

- Engouement pour la programmation **visuelle**;
- Permet d’initier différents types de public à la **programmation**.

Site: https://scratch.mit.edu/



Source de l’image: ici

Langages multiparadigmes

- Python et JavaScript : langages très populaires supportant les paradigmes impératif, fonctionnel et orienté objet.
- Oz : exemple **académique** mêlant les paradigmes logique, fonctionnel, impératif, orienté objet, contraint, distribué et concurrent.

But du cours: vous familiariser avec les paradigmes **fonctionnel** et **logique**.

Table des matières

1. Présentation du cours

2. Paradigmes de programmation

3. Rappels mathématiques

4. Le langage Haskell

5. Fonctions et types

6. Listes et tuples

Ensemble

- Un **ensemble** est une **collection** d'objets, appelés **éléments**;
- Plusieurs façons de **définir** un ensemble:
  - En **extension**;
  - En **compréhension**;
  - En **français**.
- Plusieurs **opérations** possibles: union ($\cup$), intersection ($\cap$), différence ($-$), différence symétrique ($\oplus$), produit cartésien ($\times$), etc.
- Plusieurs **relations** possibles: appartenance ($\in$), inclusion ($\subseteq$), etc.

Définition en extension

$$
\begin{array}{rcl}
C & = & \{rouge, vert, bleu\} \\
R & = & \{(1,1),(2,2),(3,3),\ldots\} \\
\mathbb{N} & = & \{0,1,2,\ldots\} \\
\mathbb{N}^* & = & \{1,2,3,\ldots\} \\
\mathbb{Z} & = & \{\ldots,-2,-1,0,1,2,\ldots\} \\
\mathbb{Q} & = & \left\{0,\frac{1}{2},\frac{1}{3},-\frac{2}{5},\frac{2}{4},\ldots\right\} \\
\mathbb{R} & = & \left\{0,1,\pi,e,-1,-\frac{1}{3},\ldots\right\} \\
\texttt{Bool} & = & \{\texttt{True},\texttt{False}\} \\
\texttt{Char} & = & \{\ldots,\texttt{'0'},\texttt{'1'},\ldots,\texttt{'9'},\ldots,\texttt{'A'},\ldots,\texttt{'Z'},\ldots\} \\
\texttt{String} & = & \{\texttt{""},\texttt{"a"},\texttt{"pomme"},\ldots\}
\end{array}
$$

Définition en compréhension

$$\begin{array}{rcl}
C & = & \{(r,g,b) \in \mathbb{N}^3 \mid 0 \leq r,g,b \leq 255\} \\
\mathbb{Q} & = & \left\{ \dfrac{a}{b} \;\middle|\; a \in \mathbb{Z}, b \in \mathbb{Z}^* \right\} \\
\mathbb{R}_+ & = & \{x \in \mathbb{R} \mid x \geq 0\} \\
[a,b] & = & \{x \in \mathbb{R} \mid a \leq x \leq b\} \\
R(x,y,w,h) & = & \{(z,t) \mid x \leq z \leq x+w, y \leq t \leq y+h\} \\
B(x,y,r) & = & \{(z,t) \mid (z-x)^2 + (t-y)^2 \leq r^2\} \\
\texttt{Char} & = & \{c \mid c \text{ est un caractère unicode}\} \\
\texttt{String} & = & \{s_1 s_2 \cdots s_n \mid n \in \mathbb{N}, s_i \in \texttt{Char} \text{ pour } i = 1,2,\ldots,n\}
\end{array}$$

Définition textuelle

- Soit $A = \{0, 1\}$. Alors
  - A^n est l'ensemble des chaînes binaires de longueur n;
  - A^* est l'ensemble des chaînes binaires (finies).
- L'ensemble $\mathcal{L}_\mathbb{Z}$ de toutes les listes d'entiers;
- L'ensemble $\mathcal{L}_\mathbb{R}$ de toutes les listes de nombres réels;
- L'ensemble C de toutes les couleurs;
- L'ensemble P de toutes les pages web;
- L'ensemble F de tous les comptes Facebook;
- etc.

Ensembles et types

- Tout **type de données** peut être vu comme un **ensemble**;
- Une **variable** v de type T signifie simplement que $v \in T$;
- Par exemple, en **Java**:
  - **`int`** est l'ensemble des entiers (signés) représentables sur 32 bits;
  - **`double`** est l'ensemble des flottants (signés) représentables sur 64 bits;
  - **`Object`** est l'ensemble de tous les objets;
  - **`ArrayList<Integer>`** est l'ensemble de toutes les listes-tableau d'entiers;
  - etc.

Fonction

- Une **fonction** est un **triplet** (A, B, G), où

  (i) A est un ensemble (fini ou infini), appelé **domaine**;
  (ii) B est un ensemble (fini ou infini), appelé **codomaine**;
  (iii) $G \subseteq A \times B$ est une relation, appelée **graphe de f** telle que pour tout $a \in A$, il existe un unique $b \in B$ tel que $(a, b) \in G$.

- On utilise la **notation**:

$$\begin{array}{rccc} f & : & A & \rightarrow & B \\ & & a & \mapsto & f(a) \end{array}$$

- L'expression $a \mapsto f(a)$ est aussi appelé **règle de correspondance**.

Exemples

$$
\begin{array}{rcccl}
\texttt{carre} & : & \text{Naturel} & \to & \text{Naturel} \\
 & & x & \mapsto & x * x \\
 & & & & \\
(+) & : & \text{Reel} \times \text{Reel} & \to & \text{Reel} \\
 & & (x, y) & \mapsto & x + y \\
 & & & & \\
\texttt{length} & : & \text{Liste} & \to & \text{Naturel} \\
 & & L & \mapsto & \text{nombre d'éléments dans } L \\
\texttt{chiffre} & : & \text{Char} & \to & \text{Booléen} \\
 & & c & \mapsto & \begin{cases} \mathit{vrai}, & \text{si } c \text{ est un chiffre.} \\ \mathit{faux}, & \text{sinon.} \end{cases} \\
\texttt{triée} & : & \text{Liste} & \to & \text{Liste} \\
 & & L & \mapsto & L' \text{ (la liste } L \text{ après tri)}
\end{array}
$$

Ensembles de fonctions

- Un ensemble peut même contenir des **fonctions**:

$$\begin{array}{rcl} \mathcal{F}_{\mathbb{N}\to\mathbb{N}} & = & \{f \mid f : \mathbb{N} \to \mathbb{N}\} \\ \mathcal{F}_{\text{Char}\to\text{Booléen}} & = & \text{l'ensemble des fonctions} \\ & & \text{de Char vers Booléen} \end{array}$$

- Un ensemble de fonctions peut aussi être vu comme un **type**.
- Nous allons manipuler ces **types** tout au long de la session.
- Exemple: la fonction **`map`**:

$$\begin{array}{rccl} \texttt{map} & : & \mathcal{F}_{\mathcal{A}\to\mathcal{B}} \times \mathcal{L}_{\mathcal{A}} & \to & \mathcal{L}_{\mathcal{B}} \\ & & (f, L) & \mapsto & [f(L[i]) \mid i = 0, 1, \ldots, |L| - 1] \end{array}$$

- Plus concrètement,

$$\texttt{map}(\texttt{carre}, [1, 10, 2]) = [1, 100, 4].$$

Fonctions récursives

- La fonction

$$\texttt{numLeafs} \quad : \quad \text{Arbre} \quad \rightarrow \quad \text{Naturel}$$

  a pour règle de **correspondance**

$$\texttt{numLeafs}(T) = \begin{cases} 0, & \text{si } T \text{ est vide;} \\ 1, & \text{si } T.\texttt{gauche} \text{ et } T.\texttt{droit} \text{ sont vides;} \\ \texttt{numLeafs}(T.\texttt{gauche}) & \\ +\texttt{numLeafs}(T.\texttt{droit}), & \text{sinon.} \end{cases}$$

- La **récursivité** est fondamentale en **informatique**, en particulier en programmations **fonctionnelle** et **logique**.

Table des matières

1. Présentation du cours
2. Paradigmes de programmation
3. Rappels mathématiques
4. Le langage Haskell
5. Fonctions et types
6. Listes et tuples

Historique

- Création de **Miranda**, en 1985, un langage **fonctionnel**, à évaluation **paresseuse**;
- Devient rapidement très **populaire**. Problème? Modèle **propriétaire**.
- En **1987**, à Portland, Oregon, formation d'un comité dont le but est d'établir un **standard**, basé sur les **points forts** des langages fonctionnels de l'époque.
- **1990**: Haskell 1.0, en l'honneur du logicien **Haskell Curry**.
- **1998**: Haskell 98, premier standard **durable**.
- **2010**: Révision **majeure** du standard.
- **Aujourd'hui**: Révisions **annuelles**.

Caractéristiques

- Basé sur le **lambda calcul**;
- Langage **interprétable**;
- Langage **compilable**;
- **Purement** fonctionnel;
- Typage **fort** et **statique**;
- Évaluation **paresseuse**;
- Permet une grande **généricité**;
- Très **peu verbeux**, par opposition à C/C++/Java.
- Très **efficace** (compilé en C).

Outils

- **Compilateurs**: GHC (utilisé en classe), Hugs (développement arrêté);
- **Documentation**: Haddock;
- **Installation/Distribution**: Cabal (analogue à EasyInstall/Pip en Python et Gem en Ruby).
- **Archive de paquets**: Hackage.

Logiciels populaires

Plusieurs logiciels ont été développés en Haskell:

- Sigma: Pour lutter contre le *spam* (Facebook);
- Pandoc: Conversion de documents;
- Xmonad: Gestion des fenêtres;
- Darcs: Système de contrôle de version (comme Git);
- etc.

Interpréteur

- Il suffit d'installer GHC;
- Puis dans un terminal, on entre `ghci`:

```
 [macos] blondin_al [~]
 $ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 1 + 2
3
Prelude> "Hello, " ++ "world!"
"Hello, world!"
Prelude>
```

- Ensuite, on saisit des **instructions** après l'invite `Prelude>`.
- Une **affectation** est faite à l'aide du mot réservé **`let`**.

```
Prelude> let l = [[1,2],[3,4]]
Prelude> l
[[1,2],[3,4]]
```

Compilateur

- Il est également possible de **compiler** un programme:
- Considérons le **programme** suivant:

```haskell
-- Fichier Hello.hs
main = putStrLn "Hello, world!"
```

- Les lignes commençant par **--** sont des **commentaires**;
- Le nom du fichier devrait commencer par une **majuscule**: préférer **H**`ello.hs` à **h**`ello.hs`;
- L'extension est **.hs**;
- On **compile** (comme avec **gcc**!):

```
ghc -o hello Hello.hs
```

- Puis on l'**exécute**:

```
./hello
```

Chargement d’un module

- Il suffit d’utiliser l’**instruction**

```
:l <Fichier.hs>
```

- Prenons le fichier **Fibo.hs** suivant:

```haskell
-- Fibonacci (inefficace)
fibo :: Integer -> Integer
fibo 0 = 0
fibo 1 = 1
fibo n = fibo (n - 1) + fibo (n - 2)
```

```
 $ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l Fibo.hs
[1 of 1] Compiling Main             ( Fibo.hs, interpreted )
Ok, modules loaded: Main.
*Main> fibo 13
233
*Main> [fibo n | n <- [0..20]]
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765]
*Main>
```

Table des matières

1. Présentation du cours
2. Paradigmes de programmation
3. Rappels mathématiques
4. Le langage Haskell
5. Fonctions et types
6. Listes et tuples

Fonctions

- Dans le **lambda-calcul**, tout est une **fonction** (même les constantes);
- La **syntaxe** en Haskell est assez proche de celle des **mathématiques**.
- Fonction **mathématique**:

$$\begin{array}{rcl} |\cdot| : \text{Reel} & \to & \text{Reel} \\ r & \mapsto & \begin{cases} r, & \text{si } r \geq 0; \\ -r, & \text{sinon.} \end{cases} \end{array}$$

- En **Haskell**:

```
valeurAbsolue :: Double -> Double
valeurAbsolue r
    | r >= 0    = r
    | otherwise = -r
```

Déclaration d'une fonction

- Lorsqu'on **déclare** une fonction, la **signature** est très importante:

```
estPositif :: [Int] -> Bool
translate :: (Float,Float) -> (Float,Float)
somme :: Int -> Int -> Int
map :: (a -> b) -> [a] -> [b]
```

- Parfois, elle peut être **omise**: le compilateur peut **déduire** le type, s'il n'y a pas d'**ambiguïté**.

```
Prelude> let somme a b = a + b
Prelude> somme 3 8
11
Prelude> :t somme
somme :: Num a => a -> a -> a
```

- **Note:** `Num` est la **classe** numérique (très différent des classes en **orienté-objet**).

Règle de correspondance

- Il y a plusieurs façons de **définir** la **règle de correspondance**.
  - Par **filtrage** (*pattern matching*);
  - Avec des **gardes**;
  - À l'aide d'**autres fonctions**;
  - En utilisant des **variables/fonctions locales**, à l'aide de `where`, `let`/`in`, `case`, etc.
- En informatique, **définir** une fonction = **implémenter** une fonction:

Appeler/appliquer une fonction

- En **mathématiques**, on utilise la notation **fonctionnelle** (avec des **parenthèses**).
- Par exemple, considérons la **translation** de vecteur (a, b):

$$T_{a,b} \;:\; \begin{array}{ccc} \mathbb{R}^2 & \to & \mathbb{R}^2 \\ (x, y) & \mapsto & (x + a, y + b) \end{array}$$

- Alors on écrit $T_{3,-4}(1, 0)$ pour dénoter le couple obtenu par translation de vecteur $(3, -4)$ du point $(1, 0)$.
- En Haskell, on utilise plutôt des **espaces**:

```
-- Translation de vecteur (a,b)
translate :: (Float,Float) -> (Float,Float) -> (Float,Float)
translate (a,b) (x,y) = (x + a, y + b)

-- Application partielle d'une fonction
translate3m4 :: (Float,Float) -> (Float,Float)
translate3m4 = translate (3,-4)
```

```
*Main> translate (3,-4) (1,0)
(4.0,-4.0)
```

Table des matières

1. Présentation du cours

2. Paradigmes de programmation

3. Rappels mathématiques

4. Le langage Haskell

5. Fonctions et types

6. Listes et tuples

Structures de données de base

- En Haskell, une structure de données fondamentale est la **liste**;
- Elle est identifiée par des **crochets** : `[1,4,2]`;
- Une **chaîne de caractères** est une liste :

```
Prelude> ['o', 'u', 'i'] == "oui"
True
Prelude> :t "oui"
"oui" :: [Char]
```

- Plusieurs **fonctions** sont disponibles pour manipuler les listes;
- En particulier, on en trouve dans le module `Data.List`:

```
Prelude> import Data.List
Prelude Data.List>
```

Homogénéité des listes

- Les éléments d'une liste doivent être de **même type**:

```
Prelude> [1,'a']
<interactive>:3:2:
    No instance for (Num Char) arising from the
        literal '1'
    In the expression: 1
    In the expression: [1, 'a']
    In an equation for 'it': it = [1, 'a']
```

- Utiliser des **tuples** ou des **enregistrement** pour des éléments de type **différent**.

Autres opérations (1/2)

```
Prelude> [1,2] ++ [3,4,5] -- Concatenation
[1,2,3,4,5]
Prelude> 'a':"llo"        -- Cons
"allo"
Prelude> head "paradigme" -- Tete
'p'
Prelude> tail "paradigme" -- Reste
"aradigme"
Prelude> last "paradigme" -- Dernier element
'e'
Prelude> init "paradigme" -- Partie initiale
```

Autres opérations (2/2)

```
Prelude> length "paradigme"        -- Longueur
9
Prelude> reverse "paradigme"       -- Renversement
"emgidarap"
Prelude> take 4 "paradigme"        -- Prefixe
"para"
Prelude> drop 3 "paradigme"        -- Suffixe
"adigme"
Prelude> maximum "paradigme"       -- Element maximum
'r'
Prelude> elem 'i' "paradigme"      -- Appartenance
True
Prelude> 'i' `elem` "paradigme"    -- Notation infixe
True
Prelude> replicate 10 'e'          -- Repetition
"eeeeeeeeee"
```

Intervalles

```
Prelude> [1..8]
[1,2,3,4,5,6,7,8]
Prelude> [3,6..20]
[3,6,9,12,15,18]
Prelude> [3,6..18]
[3,6,9,12,15,18]
Prelude> ['m'..'r']
"mnopqr"
```

Listes infinies

- Étant un langage **paresseux**, on peut facilement représenter des objets **infinis**;
- Généralement, les listes infinies sont **combinées** avec des opérations qui les rendent éventuellement finies lors de l'**évaluation**:

```
Prelude> cycle [1,2,3]
[1,2,3,1,2,3,1,2,3,1,2,3,...
Prelude> repeat 8
[8,8,8,8,8,8,8,8,8,8,8,8,...
Prelude> take 4 (repeat 'a')
"aaaa"
```

Tuples

- On peut facilement **regrouper** des données (homogènes ou non) à l'aide de **tuples**:

```
Prelude> (3,4)
(3,4)
Prelude> (1,'a')
(1,'a')
Prelude> ("Porrier", "Paradigmes", 2024)
("Porrier","Paradigmes",2024)
```

- Une fonction très **pratique**, `zip`:

```
Prelude> zip "alpha" [1..5]
[('a',1),('l',2),('p',3),('h',4),('a',5)]
Prelude> zip "fromage" (cycle [1,2,3])
[('f',1),('r',2),('o',3),('m',1),('a',2),('g',3),('e
    ',1)]
```

Compréhension de liste

- On peut construire des listes en **compréhension** (comme en **mathématiques**!):

```
Prelude> [i * i | i <- [0..10]]
[0,1,4,9,16,25,36,49,64,81,100]
Prelude> [replicate i 'a' | i <- [0..5]]
["","a","aa","aaa","aaaa","aaaaa"]
Prelude> [i * i | i <- [0..10], odd i]
[1,9,25,49,81]
```

- On peut utiliser **plusieurs** variables, même les **imbriquer**:

```
Prelude> [i * j | i <- [0..5], j <- [i..5]]
[0,0,0,0,0,0,1,2,3,4,5,4,6,8,10,9,12,15,16,20,25]
Prelude> [[i * j | i <- [0..4]] | j <- [0..3]]
[[0,0,0,0,0],[0,1,2,3,4],[0,2,4,6,8],[0,3,6,9,12]]
Prelude> [[i * j | j <- [0..3]] | i <- [0..4]]
[[0,0,0,0],[0,1,2,3],[0,2,4,6],[0,3,6,9],[0,4,8,12]]
```

Lecture et crédit

- Chaque semaine, une lecture est **proposée**;
- Elle est **complémentaire** au contenu vu en classe;
- Cours d'**aujourd'hui**: sections 1 à 3 de Learn You a Haskell for Great Good!.

Cours adapté de celui d'Alexandre Blondin Massé (UQAM), avec son aimable autorisation.