Programmation fonctionnelle

Éric Jacoboni

21 mars 2022

Université Jean Jaurès, Toulouse

Sommaire

Sommaire

Présentation du cours

Pourquoi ce cours ?

- Pour montrer une autre approche de résolution des problèmes.
- Pour augmenter votre « culture informatique ».
- Parce que cela fera de vous de « meilleurs programmeurs ».
- Parce qu'un bon développeur doit connaître le plus d'outils possibles.
- Parce que la programmation fonctionnelle, c'est rafraîchissant et généralement inconnu des cursus « professionnalisants ».
- Parce que vous aurez l'impression de devenir « malins » au lieu d'être de simples producteurs de code.
- Parce que la programmation fonctionnelle a des atouts non négligeables (programmation concurrente, simplicité du code, notamment).

Pourquoi ce cours ?

- Vous constaterez que la programmation fonctionnelle demande moins de « techniques de programmation » que la programmation impérative.
- Par contre, elle demande plus de réflexion sur le problème proprement dit : elle est donc moins « mécanique » et plus « intellectuelle ».
- Là ou la programmation impérative demande simplement de la *technique* et de la *pratique* (ce qui est à la portée du premier étudiant **sérieux** venu), la programmation fonctionnelle demande surtout de la *réflexion* (ce qui est quand même plus valorisant...).
- Quasiment tous les langages modernes (pas Go...) intègrent la programmation fonctionnelle à différents degrés.
- Les patrons de programmation enseignés ici sont pour la plupart transposables dans les autres langages (moyennant certains efforts).

Pourquoi ce cours ?

John Carmack (*Doom, Quake, Id Software, Oculus VR, ...*) :

> *Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.*

Pourquoi Haskell ?

- Ce cours n'est pas un cours sur Haskell, mais sur la programmation fonctionnelle !
- Haskell est un *langage fonctionnel pur*, ce qui permet de mieux se concentrer sur les concepts qui nous intéressent.
- Haskell est très différent des langages que vous connaissez déjà, ce qui vous forcera d'autant plus à repenser différemment.
- Et puis... parce que j'aime Haskell...

Remarque

- Vous vous demanderez sûrement pourquoi apprendre un langage que vous ne rencontrerez probablement jamais en entreprise.
- C'est parce que ce cours est *un cours sur la programmation fonctionnelle* et les langages impératifs (Java, C#, C++, etc.) ne sont pas conçus pour cela (même si on retrouve parfois certains concepts).
- Même si vous ne programmerez probablement jamais en Haskell, les concepts de ce cours vous seront utiles puisqu'ils sont repris dans quasiment tous les langages modernes (Scala, Kotlin, Rust...).

Aller plus loin...

- Nous n'aborderons pas ici les fondements théoriques de la programmation fonctionnelle (le lambda-calcul, notamment).
- Une foule de livres et d'articles sont disponibles (en anglais) pour ceux qui veulent en apprendre un peu plus.
- Si vous avez le temps, intéressez-vous aussi à Scala ou à Erlang (*Programmer en Erlang* existe en français...), à Elixir, à Lisp, à Clojure, à F#, à OCaml...
- Un développeur qui n'apprend pas constamment de nouveaux langages ou de nouveaux concepts aura toujours un train de retard...
- Profitez de vos études pour élargir votre horizon... Vous en aurez peut-être moins l'occasion plus tard.

Livres sur Haskell

- *Haskell : The Craft of Functionnal Programming*, par Simon Thompson (Addison Wesley).
- *Real World Haskell*, par O'Sullivan, Goerzen et Stewart (O'Reilly).
- *The Haskell School of Expression*, par Paul Hudak (Cambridge).
- *Programming in Haskell*, par Greg Hutton (Cambridge).
- *Learn You a Haskell for Great Good*, par Miran Lipovaca (No Starch Press).
- *Effective Haskell*, par Rebecca Skinner – en cours de publication – (The Pragmatic Programmers).
- Tout bon livre sur la programmation fonctionnelle... (*Structure and Interpretation of Computer Programs* est disponible en ligne sur le site du MIT)
- Le site officiel du langage : `www.haskell.org`...

Introduction

Historique

- Même les ordinateurs les plus récents sont construits comme leurs ancêtres : assemblages de circuits (registres, mémoire, additionneurs, décodeurs, etc.).
- Cette architecture convenait très bien aux premières applications des années 40 : tables numériques, calculs de trajectoires, déchiffrages de messages, recherche nucléaire...
- Architecture Von Neumann (qui découle de la machine de Türing) : les machines exécutent les programmes *en séquence*.
- Les programmes sont des *suites d'instructions* stockées en mémoire centrale. Ils manipulent des données de types bien connus.

Historique des langages impératifs

- Au début était le binaire... `010011101001`...
- Utilisation de *mnémoniques* pour les instructions (assembleur) et pour les emplacements mémoire (*variables nommées*).
- Utilisation de *langages évolués* : utilisation d'un *compilateur* pour traduire une instruction du langage en plusieurs instructions machine (Fortran, etc.) – 1956.
- Langages *normalisés* pour plus de portabilité (Algol, Cobol, etc.) – 1960.
- Possibilité de créer ses propres types de données à partir de types de base et de constructeurs : *tableaux*, *pointeurs*, *enregistrements*, etc. (PL/1, Algo/W, C, ...) – 1970.
- *Discipline imposée aux programmeurs* : programmation structurée, algorithmes, raffinages successifs (Pascal) – 1970.
- Incitation à la *réutilisation* de modules déjà écrits, constitution de bibliothèques et mécanismes de modularisation intégrés aux langages : Modula (78), Ada (83), C++ (90), Eiffel (94), Java (94), C# (2001), Rust (2006), Go (2009), etc. On en est là...

Programmation impérative

- Malgré toute cette évolution (en moins d'un demi-siècle) qui a mené de la programmation binaire à la programmation modulaire, le principe consiste toujours à *décrire la liste complète des actions* à réaliser pour obtenir le résultat voulu – les fameux « algorithmes » – puis à les coder.
- On s'occupe du **comment**. Un programme est une recette.
- *Programmation impérative* = programmes séquentiels + affectation (modification) des variables.
- C'est ce qu'on vous a appris depuis le début...

Programmation déclarative

- Parallèlement à cette évolution, une autre catégorie de langages s'est développée en s'inspirant des travaux de Church : *les langages fonctionnels*.
- On indique la nature des données dont on dispose et des résultats qu'on veut obtenir.
- On s'intéresse au **Quoi**. Un programme est un calcul.
- *Programmation fonctionnelle* : on calcule le résultat à partir des données, comme en mathématiques : $f(\texttt{donnees}) = \texttt{resultat}$.
- Les fonctions sont des **valeurs** comme les autres...

Langages fonctionnels

- *Lisp* (Mc Carthy, fin des années 50), *Scheme* (1974).
- *ML* (Univ. Edimbourg, 1974), *SML*.
- *HOPE* (1980), *Miranda* (1985), *Erlang* (1986), *Clean* (1987), *Haskell* (1990).
- *CAML* (Inria, 1986), *F#* (Microsoft, 2002, inspiré de *CAML*).
- Nouveaux langages tournant sur la JVM ou .NET : *Groovy* et *Scala* (2003), *Fantom* (2007), *Clojure* (2007), *Kotlin* (2011)...
- Les langages récents (*Ruby*, *Python*, *Rust*, *Swift*, *Scala*, *Kotlin*, etc.) intègrent de nombreux concepts fonctionnels, notamment l'immutabilité.

Langages fonctionnels

- La programmation fonctionnelle demandant plus de ressources (mémoire et CPU), l'industrie a massivement choisi les langages impératifs. On en est venu à penser que c'était LA façon de programmer.
- Mais, avec l'évolution du matériel et les nouveaux problèmes posés par la programmation multi-threads, la programmation fonctionnelle revient sur la scène car elle permet d'exprimer les problèmes plus clairement que les langages impératifs.
- Même *Java* s'y est mis !.
- Mais les développeurs formés à la programmation impérative négligent souvent ces mécanismes...
- ... et se privent donc de solutions simples à leurs problèmes.

Points forts

- Valeurs « immutables » :
  - plusieurs threads peuvent y accéder simultanément sans nécessiter de verrous.
  - optimisation possible du stockage en mémoire.
- Pas d'« effets de bord » :
  - fonctions plus simples à tester.
  - optimisations du code par le compilateur plus faciles et plus poussées.

Exemples

Approche impérative pour calculer la somme des n premiers entiers

```
func somme(n int) int {
  resultat := 0
  for nb := 1; nb <= n; nb++ {
    resultat += nb
  }
  return resultat
}
```

Approche fonctionnelle pour calculer la somme des n premiers entiers

```
somme :: Int -> Int
somme n = foldl (+) 0 [1..n]
```

Ou, tout simplement...

```
somme :: Int -> Int
somme n = sum [1..n]
```

Exemples

Approche impérative pour filtrer une liste

```
func pairs(liste []int) []int {
  res := []int{}
  for _, elt := range liste {
    if elt % 2 == 0 {
      res = append(res, elt)
    }
  return res
}
```

Approche fonctionnelle pour filtrer une liste

```
pairs :: [Int] -> [Int]
pairs liste = [x | x <- liste, x `mod` 2 == 0]
```

Ou...

```
pairs :: [Int] -> [Int]
pairs = filter even
```

Intérêt

- Les programmes fonctionnels ont une *nature mathématique*.
- Les fonctions décrivent l'obtention des résultats (sorties) à partir des données (entrées), *indépendamment* de l'environnement de leur utilisation $\Longrightarrow$ *Transparence référentielle* :
  - Une fonction comme *sin(x)* est référentiellement transparente car elle donnera toujours le même résultat pour un même x donné...
  - Les opérations arithmétiques sont référentiellement transparentes : 2×21 pourra toujours être remplacé par 42...
  - Mais... Une instruction comme $x{+}{+}$ n'est pas transparente car elle modifie la valeur de x : à l'appel suivant, elle ne produira donc pas la même valeur.
  - Un appel à *getchar()* n'est pas transparent car sa valeur dépend de ce qui a été saisi au clavier...
  - L'instruction *printf("Hello World")* n'est pas transparente car remplacer cet appel par sa valeur changera le comportement du programme.
  - Une fonction comme *random()* n'est pas transparente car, par définition, elle ne renvoie jamais la même valeur...

Remarques

- Rien n'empêche d'utiliser un langage impératif (comme C, Java, C# ou Python) pour écrire un programme fonctionnel : mais ce sera moins simple et le langage ne fournira pas naturellement toutes les structures de base.
- On peut mélanger les deux approches : certains problèmes s'expriment plus efficacement de façon impérative, d'autres plus naturellement de façon fonctionnelle.
- Certains langages, comme OCaml ou Scala, fournissent à la fois les structures impératives et toutes les structures fonctionnelles. Python et Rust fournissent également un certain nombre de structures fonctionnelles.
- Haskell, en revanche, est un *langage fonctionnel pur* (mais on peut utiliser un mécanisme particulier si l'on a besoin de gérer les effets de bord – pour les E/S, notamment).

Remarques (bis)

- Pour répondre à une demande importante de la part des développeurs, Java 8 (sorti en Mars 2014) a finalement ajouté un certain nombre de mécanismes fonctionnels...
- Les exemples itératifs précédents peuvent donc s'écrire également de façon fonctionnelle à partir de Java 8 :

```
int somme(int n) {
  // Calcul de la somme des n premiers entiers (Java 8)
  return IntStream.rangeClosed(1, n)
                  .sum();
}

List<Integer> pairs(ArrayList<Integer> liste) {
  // Renvoie les éléments pairs d'une liste d'entiers
  return liste.stream()
              .filter(e -> e % 2 == 0)
              .collect(Collectors.toList());
}
```

Programmer avec les fonctions

Composition de fonctions

- Soit la fonction *max2* qui renvoie le maximum de deux nombres :

$$max2(a,b) = \begin{cases} a & \text{si} \quad a > b \\ b & \text{si} \quad \text{sinon} \end{cases}$$

- On pourrait écrire *max3* de la façon suivante :

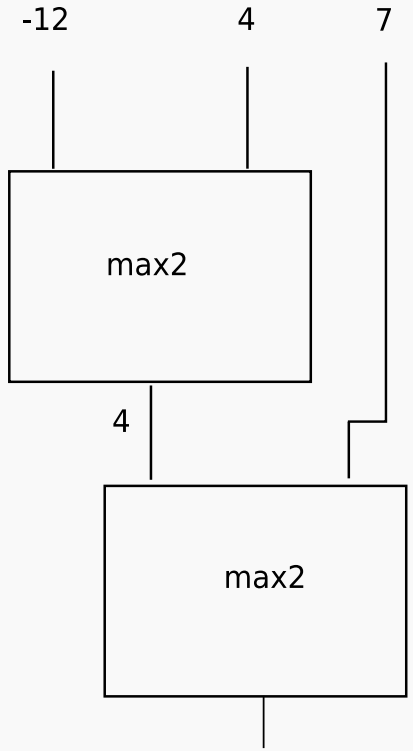
$$max3(a,b,c) = \begin{cases} a & \text{si} \quad a \geqslant b \quad \text{et} \quad a \geqslant c \\ b & \text{si} \quad b \geqslant a \quad \text{et} \quad b \geqslant c \\ c & \qquad\quad \text{sinon} \end{cases}$$

- Mais comme on sait déjà trouver le maximum de deux nombres, on peut simplement écrire :

$$max3(a,b,c) = max2(max2(a,b),c)$$

Composition de fonctions

Un appel de fonction référentiellement transparente *peut toujours être remplacé par son résultat*, ce que l'on peut représenter graphiquement par une imbrication de « boîtes noires » :



Fonctions

- Quand on aura un problème à représenter par une fonction, on décomposera ce problème en « fonctions minimales » : pour trouver le maximum de 3 valeurs, il faut bien savoir trouver le maximum de 2 valeurs $\longrightarrow$ il faut donc d'abord écrire *max2*.
- Le problème se résoudra ensuite en combinant différents appels de fonctions (*max3* se résout en termes de *max2*).
- Cette approche permet donc d'obtenir un code très *modulable* (donc facilement *testable*) et *réutilisable*.

Exemples en Go

```
// Maximum de deux entiers
func Max2(a, b int) int{
  if a > b {
    return a
  } else {
    return b
  }
}

// Maximum de trois entiers
func Max3[T](a, b, c int) int {
  return Max2(Max2(a, b), c)
}
```

```
// Signe d'un nombre
func Signe(x int) string {
  switch {
    case x < 0 : return "Négatif"
    case x == 0 : return "Nul"
    default :     return "Positif"
  }
}
```

Remarque :

La version actuelle de Go ne dispose pas de la généricité (c'est déjà disponible dans la béta...)

Exemples en Haskell

```haskell
-- Maximum de deux valeurs ordonnées
max2 x y
  | x >= y    = x
  | otherwise = y

-- Ou, plus simplement : max2 x y = if x >= y then x else y

-- Maximum de trois valeurs ordonnées
max3 x y z = max2 (max2 x y) z   -- ou : max2 x (max2 y z)
```

```haskell
-- Signe d'un nombre
signe x
  | x < 0     = "Négatif"
  | x == 0    = "Nul"
  | otherwise = "Positif"
```

Remarque :

Pour simplifier, nous n'avons pas mentionné ici les types des résultats et des paramètres des fonctions : ils seront donc ≪ inférés ≫ par Haskell (voir plus loin).

Présentation de Haskell

Présentation du langage

- Un programme Haskell est un *script* : une suite de définitions de fonctions, de types, etc.
- Une fonction Haskell peut prendre des paramètres de types quelconques et renvoyer un résultat de type quelconque.
- Une *définition de fonction* est de la forme :

```
nom_fct :: type_param_1 -> type_param_2 -> ... -> type_param_n -> type_résultat
nom_fct param_1 param_2 ... param_n = corps_fct
```

- Exemples (imparfaits...) :

```
max2 :: Int -> Int -> Int
max2 x y = if (x > y) then x else y

max3 :: Int -> Int -> Int -> Int
max3 x y z = max2 x (max2 y z)
```

Remarque :

On remarquera l'absence de parenthèses autour des paramètres...

Présentation du langage

- Un commentaire est compris entre `--` et la fin de la ligne ou entre `{-` et `-}`
- Le symbole `::` signifie « est du type ». Haskell est un langage *fortement* typé mais, dans la plupart des cas, il sait *inférer* le type (dans tous les exemples précédents, on aurait donc pu omettre la déclaration de type).
- Haskell est sensible à la casse : les noms des types, des modules et des constantes *doivent* commencer par une majuscule. Les noms des fonctions et des paramètres *doivent* commencer par une minuscule.
- Il existe également certaines *conventions de nommage* que nous présenterons au fur et à mesure.

Remarque

Pour les TP, nous utiliserons le compilateur `ghc`, qui est disponible pour toutes les plates-formes (voir `http://www.haskell.org/ghc/`). Il est également disponible en ligne sur `http://repl.it`

Principaux concepts

- Haskell est un *langage fonctionnel pur* : aucun effet de bord possible.
- Évaluation *paresseuse* : n'évalue les expressions que si cela est nécessaire (on peut donc notamment définir des listes infinies).
- *Fortement typé* (classes de types, instances de types, etc.). L'utilisateur peut créer ses propres types en plus des types prédéfinis.
- En cas de besoin, Haskell tente d'*inférer* le type d'une expression.
- *Indentation obligatoire* pour définir les blocs de code (*layout*) (comme en Python).

Types de base

- *Bool*, *Int*, *Integer*, *Word*, *Float*, *Double*, *Rational*, *Char* et *String* (liste de *Char*), etc.
- *Instances de classes* : les nombres sont des instances de la classe *Num*, les entiers sont des instances de *Integral*, *Num* est une instance de *Eq*, etc.
- Certaines opérations exigent que leurs paramètres soient des instances d'une classe particulière (la comparaison exige des instances de *Ord*, par exemple).
- En Haskell, les « classes » sont en fait des « classes de types » (un peu comme les interfaces de Java).
- Une déclaration comme `max2 :: Ord a => a -> a -> a` se lit comme : « les deux paramètres et le résultat de *max2* sont d'un type qui est une instance de *Ord* ».
- La notation `Ord a =>` est une *contrainte de type* où *a* est une *variable de type* : ici, elle désigne n'importe quel type instance de *Ord*.

Classes de types

- **Eq** : dispose de l'égalité et l'inégalité ($==$ et $/=$)
- **Ord** : dérive de *Eq*. Dispose en plus des opérateurs de comparaison classiques, des fonctions *min* et *max* et d'une fonction *compare* prenant deux *Ord* en paramètre.
- **Num** : Classe de base pour tous les nombres. Elle définit les opérations arithmétiques classiques, sauf la division et le reste, ainsi que les fonctions *negate*, *abs* et *signum*.
- **Integral** : Dérive de *Num*. Classe de base pour les entiers (*Int* et *Integer*). Définit les fonctions *quot*, *rem*, *div*, *mod*, *quotRem*, *divMod* et *toInteger*.
- **Fractional** : Dérive de *Num*. Classe de base pour *Float*, *Double*. Définit la division réelle (notée $/$).
- **Show** : Définit la fonction *show* permettant de convertir un objet en *String*. Tous les types prédéfinis dérivent de *Show*.
- **Read** : Définit la fonction *read* permettant de convertir une chaîne dans un type instance de *Read*. Tous les types prédéfinis dérivent de *Read*.

Les fonctions show et read

Exemples

```
ghci> show 42
"42"
ghci> show 3.14159
"3.14159"
ghci> show True
"True"
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "4"
erreur...
ghci> :t read
read :: (Read a) => String -> a
ghci> read "4" :: Int
4
ghci> read "4" :: Float
4.0
```

Le type Bool

- Instance de la classe *Eq* (donc égalité et différence).
- Deux valeurs : *True* et *False*.
- Opérateurs *&&* et *||* en court-circuit. Opérateur *not*.
- Pas d'opérateur ou-exclusif prédéfini (mais il est simple à écrire)...

```
exOr :: Bool -> Bool -> Bool
exOr x y = x /= y
```

Remarque

False et True sont les deux seules valeurs de Bool. Il n'y a notamment pas d'équivalence entre 0 et faux comme dans d'autres langages...

Les types Int et Integer

- Instances de *Integral* (qui est elle-même une instance de *Num*), de *Ord* (donc de *Eq*) et de *Enum*.
- *Int* permet de représenter des valeurs *entières de taille finie* (généralement 32 ou 64 bits). *Integer* est identique à *Int*, mais ses valeurs ne sont pas finies (elles dépendent de la taille mémoire).
- Il existe également les types *Int8*, *Int16*, *Int32* et *Int64* pour les entiers signés et les types *Word*, *Word8*, etc. pour les entiers non signés.
- Opérateurs *+*, -, ***, *div*, *mod*, *rem*, *abs*, ^ (puissance entière positive ou nulle), *signum* (renvoie -1, 0 ou 1).
- En Haskell, tous les opérateurs uniquement composés de symboles sont *infixes* : ils se notent donc *x `OP` y*. Exemple : *4 + 2*, *6 * 3*.
- Les opérateurs alphabétiques sont des opérateurs *préfixes*, qui se notent donc *`OP` x y*. Exemple : *div 4 2*, *rem 3 1*.

Opérateurs infixes et préfixes

- Tout opérateur préfixe (*div*, par exemple) peut s'écrire de façon infixe en l'entourant d'apostrophes inverses : *4 `div` 2*.
- Tout opérateur infixe (*+*, par exemple) peut s'écrire de façon préfixe en l'entourant de parenthèses : *(+) 4 2*. *Nous verrons plus loin que pour passer un opérateur en paramètre à une fonction, il faut le mettre en préfixe....*
- Attention au moins unaire : *signum -12* est lu comme la soustraction de *signum* et de 12... Il faut donc toujours mettre un nombre négatif entre parenthèses : *signum (-12)*.

Les types Float et Double

- Dérivent de *Fractional* (qui est lui-même une instance de *Num*) et de *Ord* (donc de *Eq*).
- Incompatibles avec les entiers (et réciproquement) : il faut passer par une *conversion explicite*.
- Opérateurs +, -, *, /, *abs*, ^ (puissance entière positive ou nulle), ** (puissance réelle), *signum* (renvoie -1.0, 0.0 ou 1.0).
- *Float* et *Double* dérivent également de *Floating*, ce qui permet de leur appliquer les fonctions mathématiques usuelles : *sin*, *cos*, *tan*, etc. (*Floating* définit également la valeur *pi*).
- *Float* et *Double* dérivent aussi de *Enum*, ce qui permet notamment d'utiliser les fonctions *succ* et *pred* sur les flottants.

Remarque

Il est conseillé d'utiliser Double plutôt que Float car les opérations sur les Double sont bien plus optimisées que celles sur les Float.

Conversions flottants vers entiers

- Les fonctions *ceiling*, *floor*, *round* et *truncate* s'appliquent à un flottant et renvoient un *Integral*.
- La fonction *properFraction* décompose un flottant en partie entière et partie fractionnaire

Flottant $\longrightarrow$ entier

```
ghci> 5.6 + 4
9.6         OK car les littéraux 4 et 5.6 a été considérés comme des Num
ghci> val = 4
ghci> :t val
val :: Num p => p
ghci> floor 5.6 + val
9
ghci> ceiling 5.6 + val
10
ghci> round 5.6 + val
10
ghci> truncate 5.9
5
ghci> properFraction 5.9
(5,0.9000000000000004)
```

Conversions entiers vers flottants

La fonction *fromIntegral* s'applique à un entier et renvoie un *Num* :

Entier $\longrightarrow$ flottant

```
ghci> floor 5.6 + 4.8
Erreur : on ne peut pas additionner un Integral (floor 5.6) et un Num (4.8)

ghci> fromIntegral (floor 5.6) + 4.8
9.8
```

Le type Char

- Dérive de *Ord* et de *Enum*. Permet de représenter les caractères (lettres, chiffres, caractères spéciaux). Un caractère est codé en Unicode.
- Représenté entre *apostrophes simples*, comme en C.
- Utilisation des *séquences d'échappement* de C (`\n`, `\t`, `\'`, `\"`, etc.).
- Utilisation de codes en décimal ou hexadécimal (`'\97'` et `'\x61'` représentent tous les deux la lettre `'a'`).
- Les fonctions du module *Data.Char* fournissent des opérations supplémentaires.
- Le type *String* est un synonyme de « liste de *Char* ».

Le module Data.Char

Utilisation du module Char

```
ghci> Data.Char.toUpper 'a'
'A'
ghci> import Data.Char
gchi> toUpper 'a'
'A'
ghci> :t toUpper
toUpper :: Char -> Char
ghci> :doc toUpper
Convert a letter to the corresponding upper-case letter, if any.
Any other character is returned unchanged.
ghci> isDigit 'a'
False
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> :browse Data.Char
(...)
chr :: Int -> Char
digitToInt :: Char -> Int
intToDigit :: Int -> Char
isAlpha :: Char -> Bool
isAlphaNum :: Char -> Bool
(...)
```

Format d'un script

- Haskell utilise l'indentation pour délimiter les blocs.
- Un bloc se termine par la première ligne indentée comme la ligne qui débute le bloc.
- Placez des lignes blanches entre les définitions pour améliorer la lisibilité.
- Généralement, les fichiers source Haskell portent l'extension *.hs*

Structures de choix

Exemple de if (le else est obligatoire)

```
max2 :: Ord a => a -> a -> a
max2 x y =
  if x >= y then x else y   -- les deux parties doivent renvoyer un Ord a.
```

Exemple de garde

```
max2 :: Ord a => a -> a -> a
max2 x y
 | x >= y    = x          -- doit renvoyer un Ord a
 | otherwise = y          -- idem
```

Exemple de pattern-matching

```
ou :: Bool -> Bool -> Bool
ou True y = True
ou _ y    = y
```

Structures de choix : l'instruction case

- L'expression *case* est la structure fondamentale du pattern-matching en Haskell.
- Sa syntaxe générale est la suivante :

```
case expr of
   motif1  -> expr1
   motif2  -> expr2
   ...
```

- L'expression *case* complète produira l'expression qui correspond au premier motif capturé.
- L'écriture des fonctions par pattern-matching (comme la fonction *ou* précédente) est en réalité un simple sucre syntaxique pour simplifier l'écriture d'une expression *case* :

```
ou :: Bool -> Bool -> Bool
ou x y = case x of
  True   -> True
   _     -> y        -- Le motif '_' capture tout ce qui n'a pas déjà été capturé
```

Itérations

- Haskell étant un langage fonctionnel pur, il n'a pas de structures itératives car elles nécessitent l'utilisation de variables et leur affectation. Les répétitions sont créées par des *appels récursifs*.
- Exemple de répétition par récursivité :

```
facto :: Int -> Int
facto 0 = 1
facto n = n * facto (n - 1)
```

- Autre version de facto, avec une *garde* :

```
facto2 :: Int -> Int
facto2 n
  | n == 0  = 1
  | n > 0   = n * facto2 (n - 1)
```

- Autre version de facto, avec une liste :

```
facto3 :: Int -> Int
facto3 n = product [2..n]
```

Traitement des erreurs

- Un appel à *facto (-2)* provoquera une boucle sans fin.
- Un appel à *facto2 (-2)* provoquera une exception :

```
*** Exception: essai.hs:(5,1)-(7,32): Non-exhaustive patterns in
                                      function facto2
```

- Un appel à *facto3 (-2)* renverra 1, ce qui est faux...
- La fonction *error* permet de déclencher explicitement une erreur avec un message plus explicite :

```
facto :: Int -> Int
facto n
 | n < 0     = error "La factorielle n'est définie que sur N !"
 | otherwise = product [2..n]
```

Remarques

- Écrire une fonction susceptible d'appeler *error* revient à écrire une *fonction partielle* (qui n'est pas définie sur toutes ses entrées, contrairement à une *fonction totale)*, ce qui est une mauvaise pratique de programmation. De plus, *error* produit un *effet de bord* (un affichage...)
- L'utilisation du type *Maybe* ou d'une liste permettra d'éviter ce problème.