Introduction à la programmation fonctionnelle

https://www.cril.univ-artois.fr/~delorme/haskell

Programmation fonctionnelle

- Une autre façon d’écrire les programmes
- *Paradigme* fonctionnel
  - Procédural
  - Objet
  - …
- Très ancien : lambda-calcul, années 1930
- Langage utilisé ici : Haskell

À chaque langage son paradigme

- C : procédural
- Java : objet
- C++ : multi-paradigme
- Python : multi-paradigme
- Javascript : multi-paradigme
- ...
- Haskell : fonctionnel

Paradigme fonctionnel

- On n’écrit que des fonctions (au sens mathématique)
- Une fonction associe une valeur à une autre valeur
  - opposé : $\mathbb{R} \to \mathbb{R}$
  - inverse : $\mathbb{R}^* \to \mathbb{R}$
  - successeur : $\mathbb{N} \to \mathbb{N}$
  - plus : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - impair : $\mathbb{N} \to \mathbb{B}$
  - impair : Int → Bool
  - longueur : String → Int
  - JourDeLaSemaine : Int x Int x Int → String

Quelques exemples en Haskell...

```
opposé x = -x
inverse x = 1/x
successeur n = n + 1
plus a b = a + b
impair n = n % 2 == 1
```

`impair 3` est vrai

`successeur 3` vaut 4

`impair (successeur 3)` est faux

Propriétés importantes

- La valeur associée à une entrée donnée sera toujours la même
- Le résultat renvoyé par la fonction ne dépend *que* des paramètres d'entrée
  - l'opposé de 3 sera toujours -3
  - l'inverse de 5 sera toujours 0,2
  - Le successeur de 10 sera toujours 11
  - Ajouter 5 et 3 donnera toujours 8
  - 7 sera toujours impair
  - « toto » sera toujours de longueur 4
  - Le 9 janvier 2023 sera toujours un lundi

Propriétés importantes

- Une fonction ne modifie pas ses paramètres
- Pas de notion de variable ou d'état comme en procédural
  - Quand on calcule l'opposé de 3 on ne modifie pas 3
  - Quand on calcule l'opposé de x on ne modifie pas x
  - Quand on additionne a et b on ne modifie pas a ni b
  - Calculer la longueur de « toto » ne nécessite pas de la modifier
  - Etc.

Propriétés importantes

- On ne peut pas modifier la valeur d’une variable une fois qu’elle a été créée
  - Comment on fait pour écrire « oterVoyelles » qui enlève toutes les voyelles d’une chaîne ?
  - Comment on fait pour écrire la fonction « oterVie » qui fait perdre une vie au joueur ?
  - Comment on fait pour écrire « faireVirement » qui permet de transférer une somme d’argent d’un compte à l’autre ?

oterVie

- Python (procédural)

  Un joueur est une liste de 3 éléments : le pseudo, le score, le nombre de vies

```
def oter_vie(joueur):
    joueur[2] = joueur[2] - 1

j = ["toto", 5300, 3]
oter_vie(j)
```

j vaut maintenant ["toto", 5300, 2]

On l’a modifié !

oterVie en Haskell

On va créer une fonction oterVie : Joueur → Joueur qui à un joueur, va associer un autre joueur avec une vie en moins

```
oterVie :: (String, Int, Int) -> (String, Int, Int)
oterVie (nom, score, nbVies) = (nom, score, nbVies - 1)
j = ("toto", 5300, 4)
j2 = oterVie(j)
```

- j2 vaut ("toto", 5300, 3)
- On a créé une nouvelle variable
- j n’a pas été modifié

En Haskell...

- On ne va pas écrire oterVie qui modifie le paramètre de type joueur
  - On va renvoyer une nouvelle valeur qui représente le nouvel état du joueur
- On ne va pas écrire oterVoyelles qui modifie la chaîne passée en paramètre
  - On va renvoyer une nouvelle chaîne, sans les voyelles
  - On ferait aussi comme ça en python !
- On ne va pas écrire faireVirement qui modifie deux comptes bancaires
  - On va renvoyer deux nouvelles valeurs, une pour chaque compte

À quoi ça sert ?

- Faciliter les tests
  - Une fonction n’a pas d’*effet de bord*
  - Elle ne modifie rien
  - La sortie ne dépend que des entrées
  - À une entrée donnée, une seule sortie donnée
  - On peut tester les fonctions isolément les unes des autres
  - Si ça marche pendant le test, ça marchera toujours

À quoi ça sert ?

- Faciliter la parallélisation
  - On ne risque pas de modifier une même variable en même temps (accès concurrent)
  - Dans un programme impératif, que se passe-t-il si un thread t1 modifie l'objet o pendant que le thread t2 regarde sa valeur ?
    - Réponse : un bug, très difficile à détecter et à reproduire

Pourquoi on ne l’utilise pas partout ?

- Certaines choses difficiles à écrire en fonctionnel
  - Manipulation de matrices
  - Tri efficace sans créer de copie
  - Tout ce qui modifie le monde extérieur : hello world
  - Générer un nombre aléatoire
  - Connaître la date du jour
- Haskell permet quand même de modifier le monde
- Mais c’est compliqué

Utile ou pas, alors ?

- La plupart des langages proposent des outils fonctionnels
- Savoir utiliser le bon outil au bon moment
- Apprendre Haskell vous fera progresser en python / java / javascript / etc. et même en C

Introduction au langage Haskell

- Haskell est un langage purement fonctionnel
  - Pas de construction impérative
- Haskell est un langage compilé
  - On écrit le code source
  - On donne le code source en entrée au compilateur
  - Ce dernier produit un fichier exécutable
  - Comme en C
  - Pas comme python, PHP ou javascript qui sont interprétés
  - Mais (comme en python) il y a un interprète qui permet de tester des fragments de code sans compiler

Introduction au langage Haskell

- Haskell est un langage fortement typé
  - On ne peut pas additionner un entier et une chaîne par exemple
  - Comme Python ou Java
  - Pas comme C ou PHP ou Javascript (typage faible)
  - Haskell est très fortement typé, il a un des systèmes de types les plus poussés
- Haskell est un langage typé statiquement
  - On connaît tous les types au moment de la compilation
  - Comme C ou Java
  - Pas comme Python ou PHP ou Javascript (typage dynamique)
- Haskell fait de l'*inférence de type*
  - Il déduit les types sans qu'on aie besoin de les lui indiquer (en général)

Quelques types de base en Haskell

- Entiers

  Il en existe deux types :

  - Int : entiers de taille limitée (sur 64 bits)
  - Integer : entiers « longs », pas de taille limite, mais moins efficace

- Booléens

  Bool : True et False

  Opérateurs : &&, ||, not

- Réels

  Pas de vrai réel mais des représentations en virgule flottante

  Float (simple précision) et Double (double précision)

Quelques types de base en Haskell

- On dispose des opérateurs habituels

```
5 + (7 * 3) == 0
> False

5 >= 2 && 3 != 0
> True

10 / 2
> 5.0

1000 * 1.7e-3
> 1.7

100 * 1.7e-3
> 0.1699999999998  → Attention aux arrondis avec les flottants !
```

Quelques types de base en Haskell

- Caractère

  Char : 'a' est un Char, '0' aussi, '®' aussi, '☠' aussi

  Guillemets simples
- Chaîne de caractères

  String : "toto" est un String, "a" aussi, "" aussi

  Guillemets doubles
- 'a' et "a" ne sont pas du même type !
  - 'a' est un caractère
  - "a" est une chaîne contenant un caractère

Les listes

- Ça se complique (un peu)
- Haskell est fortement typé
  - Donc les listes sont typées
  - Tous les éléments d’une liste ont le même type
- [3, 5, 12] est une liste d’entiers (type [Int]) de taille 3
- [False] est une liste de booléens (type [Bool]) de taille 1
- ~~[3, True]~~ n’est pas une valeur valide !
- ['t', 'o', 't', 'o'] est de type [Char] et de taille 4
  - ['t', 'o', 't', 'o'] == "toto" : c’est une écriture alternative
  - Les String sont des [Char] et vice versa

Les listes

- On peut aussi avoir des listes de listes
- [[3, 5, 12], [2]] est une liste de liste d’entiers (type [[Int]]) de taille 2
- [[False, False, True]] est une liste de liste de booléens (type [[Bool]]) de taille 1
- ["toto", ['t', 'i', 't', 'i']] == ["toto", "titi"] est de type [String] ou [[Char]]
- ~~[[1, 2, 3], [False]]~~ est invalide
- ~~[[1, 2, 3], 4, 5]~~ est invalide

Les listes

- Il y a beaucoup de fonctions et d'opérateurs prédéfinis sur les listes
- Longueur d'une liste l : `length l`

```
length [1, 2, 3] == 3
```

- Accéder au ième élément de l : `l !! i`

```
[1, 2, 3] !! 1 == 2
```

  - Les indices démarrent à 0
- Accéder au premier élément de l : `head l`

```
head [1, 2, 3] == 1
```

- Accéder à tout sauf le premier élément de l : `tail l`

```
tail [1, 2, 3] == [2, 3]
```

- Prendre les n premiers éléments : `take n l`

```
take 2 [1, 2, 3] == [1, 2]
```

Les fonctions

- Élément fondamental en Haskell
- Une fonction n’exécute pas d’instruction
- Elle calcule et renvoie une valeur
- Pas besoin de return comme en python : il est implicite
- Une fonction a un type : son paramètre d’entrée et celui de sortie
  - La fonction « opposé » est de type Int -> Int
- Quand il y a plusieurs paramètres, on met plusieurs flèches (on verra plus tard pourquoi)
  - La fonction « plus » est de type Int -> Int -> Int
- Une fonction sans paramètre ? Ça n’existe pas !
  - C’est une constante

Les fonctions

- Attention à l'écriture des appels de fonction
- C'est inhabituel mais on verra la raison plus tard
- On n'écrit pas ~~f(x, y, z)~~ mais **`f x y z`** ou (**`f x y z`**)
- Pour appeler la fonction « plus » avec les paramètres a et b, on écrira :

```
plus a b
```

- Si on veut additionner l'opposé de a et l'inverse de b on fera :

```
plus (opposé a) (inverse b)
```

~~**`plus opposé a inverse b`**~~ ← **`ambigu !`**

Écrire une fonction

- Écrivons une fonction f qui, à un entier x donné, renvoie son successeur
- On veut écrire la fonction mathématique f : x → x + 1
- En Haskell, on écrira :

```
f x = x + 1
```

- On vient de définir f, et le compilateur infère (déduit) son type

```
f :: Int -> Int
```

(en vrai c'est plus compliqué que ça : pourquoi pas des Float par exemple ? Ou des Double ? on y reviendra)

Utiliser une fonction

- Une fois que f est définie, on peut l'utiliser

```
opposé (f 3)
> -4

plus 3 (f 3)
> 7

plus (f 3) (f (f 3))
> 9

[0, f 0, f (f 0), f (f (f 0))]
> [0, 1, 2, 3]
```

Les listes (rappel et typage)

- Longueur d'une liste l : `length l`

```
length [1, 2, 3, 2] == 4
length [True, True] == 2
length "toto" == length ['t', 'o', 't', 'o'] == 4


length :: [a] -> Int
```

Types génériques

- C’est quoi ce paramètre de type [a] ?
- En fait length peut s’appliquer à n’importe quel type de liste
- Il s’applique à une liste de valeur d’un type quelconque (un type *générique*)
- Par convention, on appelle le type générique : a
- Donc le paramètre est [a] (« liste de valeurs de type a, quel que soit a »)
- Cela veut dire qu’on peut remplacer a par ce qu’on veut : Int, Bool, Char, etc.

Les listes (rappel et typage)

- Longueur d'une liste l : `length l`

  ```
  length :: [a] -> Int
  ```

- Accéder au premier élément de l : `head l`

  ```
  head :: [a] -> a
  ```

- Tout sauf le premier élément de l : `tail l`

  ```
  tail :: [a] -> [a]
  ```

- Prendre les n premiers éléments : `take n l`

  ```
  take :: Int -> [a] -> [a]
  ```

- Accéder au ième élément de l : `l !! i`

  ```
  (!!) :: [a] -> Int -> a
  ```

  NB : les opérateurs sont des fonctions comme les autres ! Autre exemple :

  ```
  (+) :: Int -> Int -> Int
  ```

Écrire une fonction

- Écrivons une fonction moyenne qui calcule la moyenne entre 3 valeurs

```
moyenne :: Double -> Double -> Double -> Double
moyenne a b c = (a + b + c) / 3
```

Écrire une fonction

- Écrivons une fonction passe qui indique si la moyenne est ≥ 10

```
passe :: Double -> Double -> Double -> Bool
passe a b c = (moyenne a b c) >= 10.0
```

Les conditionnelles (if)

- Écrivons une fonction jugement qui renverra « t'es nul » si la moyenne est < 10, ou « bravo » dans le cas contraire

```
jugement :: Double -> Double -> Double -> String
jugement a b c =
   if (passe a b c)
      then "Bravo"
      else "T'est nul"
```

- En Haskell, le if n'exécute pas d'instruction, il calcule et renvoie une valeur
  - Comme les fonctions
  - Il y a forcément un else
  - Les deux branches doivent renvoyer le même type

Les tuples

- Tous les éléments d’une liste ont le même type
- Comment faire pour mettre ensemble des valeurs de types différents ?
- Les tuples sont là pour ça !
  - Un joueur est l’association d’un nom, d’un score et d’un nombre de vies
  - Un joueur est l’association d’une chaîne et de deux entiers
  - Un joueur est de type (String, Int, Int)
  - ("toto", 3500, 3) est de type (String, Int, Int)
- Les tuples ont une taille fixe
- Les listes sont de taille variable mais ont un type unique

Les énumérations

- Un booléen (type Bool) c'est une valeur parmi l'ensemble {True, False}

```
data Bool = True | False
```

- On peut créer nos propres énumérations

```
data FeuTricolore = Vert | Orange | Rouge

data Jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi |
Dimanche
```

- On a créé deux nouveaux types qu'on peut utiliser :

```
[Rouge, Rouge, Vert, Orange, Rouge] :: [FeuTricolore]

(Samedi, Rouge, "toto") :: (Jour, FeuTricolore, String)
```

Écrire une fonction

- Écrivons une fonction mention qui renverra la mention obtenue en fonction de la moyenne
- On va d'abord créer un type Mention (une énumération)

```
data Mention = Echec | Passable | AssezBien | Bien | TresBien
```

- Puis on va créer la fonction

```
mention :: Double -> Double -> Double -> Mention
mention a b c =
   if (moyenne a b c) < 10
      then Echec
      else if (moyenne a b c) < 12
         then Passable
         else if (moyenne a b c) < 14
            then AssezBien
            else if (moyenne a b c) < 16
               then Bien
               else TresBien
```

Variables en Haskell (let)

- C’est moche : on recalcule plusieurs fois la moyenne
- On va utiliser une variable pour la calculer une seule fois
- Une variable en Haskell est plutôt une constante : une fois sa valeur fixée, elle ne changera plus

```
mention :: Double -> Double -> Double -> Mention
mention a b c =
   let moy = (moyenne a b c) in
      if moy < 10
         then Echec
         else if moy < 12
            then Passable
            else if moy < 14
               then AssezBien
               else if moy < 16
                  then Bien
                  else TresBien
```

Indentation et if imbriqués

- Pour des raisons de lisibilité, on pourrait aussi écrire :

```
mention :: Double -> Double -> Double -> Mention
mention a b c =
   let moy = (moyenne a b c) in
      if moy < 10
         then Echec
      else if moy < 12
         then Passable
      else if moy < 14
         then AssezBien
      else if moy < 16
         then Bien
      else
         TresBien
```

Écrire une fonction

- Pourquoi on a créé un type Mention ? On aurait pu utiliser String, non ?

```
mentionBof :: Double -> Double -> Double -> String
mentionBof a b c =
   let moy = (moyenne a b c) in
      if moy < 10
         then "Echec"
      else if moy < 12
         then "Passable"
      else if moy < 14
         then "AssezBien"
      else if moy < 16
         then "Bien"
      else
         "TresBien"
```

Problème

- On a une liste de tuple qui associent le nom d'un étudiant et sa mention

```
l :: [(String, String)]
l = [("Toto", "Bien"), ("Titi", "ABien")]
```

- Oups ! J'ai écrit "ABien" au lieu de "AssezBien"

```
l :: [(String, Mention)]
l = [("Toto", Bien), ("Titi", ABien)]
```

- Ne compile pas : le compilateur nous a évité un bug !

Problème 2

- On a une liste de tuple qui associent le nom d'un étudiant et sa mention

```
l :: [(String, String)]
l = [("Toto", "Bien"), ("AssezBien", "Titi")]
```

- Oups ! J'ai inversé nom et mention

```
l :: [(String, Mention)]
l = [("Toto", Bien), (AssezBien, "Titi")]
```

- Ne compile pas : le compilateur nous a évité un bug !

Moralité

- Le système de types est là pour nous aider !
- Plus on est explicite, plus on définit précisément les types, plus on découvre de bugs à la compilation
- C’est difficile de faire plaisir au compilateur
- Mais une fois qu’un programme compile, on a de bonnes chances pour qu’il fonctionne !
- Attention ! Ce n’est pas parce que ça compile qu’on a la *garantie* que le programme fonctionne

Écrire une fonction

- Quel est le successeur d'un feu ?

```
Suivant :: FeuTricolore -> FeuTricolore
suivant feu =
    if feu == Rouge
        then Vert
        else if feu == Orange
            then Rouge
            else Vert
```

Pattern matching

- Permet de simplifier l'écriture des fonctions
- On peut supprimer des if
- Au moment de l'exécution, le programme va prendre la *première* branche qui peut être choisie

```
suivant :: FeuTricolore -> FeuTricolore
suivant Rouge  = Vert
suivant Orange = Rouge
suivant Vert   = Orange
```

Pattern matching

```
estWeekend :: Jour -> Bool
estWeekend Samedi    = True
estWeekend Dimanche  = True
estWeekend Lundi     = False
estWeekend Mardi     = False
estWeekend Mercredi  = False
estWeekend Jeudi     = False
estWeekend Vendredi  = False
```

Pattern matching

- On peut aussi utiliser des clauses attrape-tout avec des variables au lieu de constantes
- Au moment de l'exécution, le programme va prendre la *première* branche qui peut être choisie
- Attention à l'ordre ! Si on met la clause attrape-tout au début, elle sera toujours exécutée !

```
estWeekend :: Jour -> Bool
estWeekend Samedi   = True
estWeekend Dimanche = True
estWeekend x        = False
```

Pattern matching

- On peut aussi utiliser des clauses attrape-tout avec des variables au lieu de constantes
- Si on n’utilise pas la variable, on peut aussi utiliser le joker « _ »

```
estWeekend :: Jour -> Bool
estWeekend Samedi   = True
estWeekend Dimanche = True
estWeekend _        = False
```

If fonctionnel

- Comme les deux branches du if ont le même type, on peut utiliser un if au milieu d'une expression :

  ```
  if x > 3 then "toto" else "titi"
  ```

  est de type String
- On peut donc avoir par exemple :

  ```
  ["tata", "tutu", if x > 3 then "toto" else "titi"]
  ```

  qui est de type [String]

Les boucles

- Mais comment on fait pour les boucles ?
  - Faire une boucle suppose de surveiller l'état d'une condition

    python : « while x != 0: »

    python : « for i in range(nb_joueurs): »
  - Si les variables de la condition ne changent pas, la boucle ne se termine pas…
  - Or en Haskell, les valeurs des variables ne changent pas…
  - → Pas de while ou de for en Haskell !
- Seule solution : la récursivité !
- Récursivité = récurrence en mathématiques

Factorielle impérative

- Python

```
def fact(n):
    res = 1
    while n > 1:
        res = res * n
        n = n - 1
    return res
```

- Ce n’est pas fonctionnel : on modifie res et n

Factorielle au sens mathématique

- fact : $\mathbb{N} \rightarrow \mathbb{N}$
- fact(n) = $\begin{cases} 1 \text{ si } n = 0 \\ n \,.\, \text{fact}(n-1) \text{ sinon} \end{cases}$

- Haskell :

```
fact :: Int -> Int
fact n =
   if n == 0
      then 1
      else n * fact (n-1)
```

Factorielle récursive

- Python

```
def fact(n):
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

C’est possible aussi en python !

Pattern matching

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

La récursivité

- Ça n'est pas toujours intuitif !
- Parfois on sait comment écrire sous forme de boucle, mais pas trop comment traduire sous forme récursive
- Souvent, on va utiliser une fonction auxiliaire appelée avec les valeurs initiales
- Règle générale :

```
def f(x):
    res = valeur initiale
    while condition:
        instructions
    return res
```

```
f x = f' x init

f' x etat =
    if not condition
        then res
        else f' x (modif etat)
```

Exemple : maximum d'une liste non vide

- Python « naïf »

```
def max(l):
    res = l[0]
    for i in range(len(l)):
        if l[i] > res:
            res = l[i]
    return res
```

Haskell « naïf »

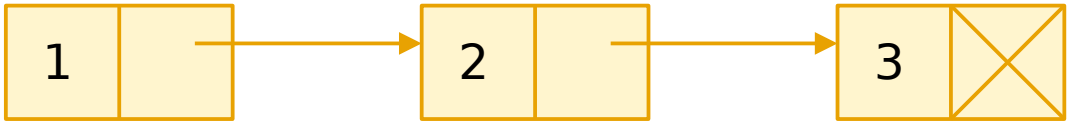
```
max :: [Int] -> Int
max l = max' l 0 (l!!0)

max' :: [Int] -> Int -> Int -> Int
max' l i res =
    if i == length l
        then res
        else if (l!!i) > res
            then max' l (i+1) (l!!i)
            else max' l (i+1) res
```

Listes chaînées

- En fait, les listes en Haskell sont des listes chaînées

```
[1, 2, 3] == 1:(2:(3:[]))
```

- L'opérateur « : » permet d'associer un élément à une liste déjà existante
- 3:[] donne [3]
- 2:[3] donne [2, 3]
- 1:[2, 3] donne [1, 2, 3]
- head donne le premier élément : `head [1,2,3] == head 1:[2,3] == 1`
- tail donne tout sauf le premier élément : `tail [1,2,3] == tail 1:[2,3] == [2,3]`



Exemple : maximum d'une liste non vide en exploitant l'aspect « liste chaînée »

- Version 1

```
max l = max' l 0 (l!!0)

max' l i res =
    if i == length l
        then res
        else if (l!!i) > res
            then max' l (i+1) (l!!i)
            else max' l (i+1) res
```

Version 2

```
max l = max' l (l!!0)

max' :: [Int] -> Int -> Int
max' l res =
    if l == []
        then res
        else if (head l) > res
            then max' (tail l) (l!!i)
            else max' (tail l) res
```

Exemple : maximum d'une liste non vide en exploitant l'aspect « liste chaînée »

- Version 2

```
max l = max' l (l!!0)

max' l res =
    if l == []
        then res
        else if (head l) > res
            then max' (tail l) (l!!i)
            else max' (tail l) res
```

Version 3

```
max l = max' l (l!!0)

max' [] res = res
max' l res  =
    if (head l) > res
        then max' (tail l) (l!!i)
        else max' (tail l) res
```

Exemple : maximum d'une liste non vide en exploitant l'aspect « liste chaînée »

- Version 3

```
max l = max' l (l!!0)

max' [] res = res
max' l res  =
    if (head l) > res
        then max' (tail l) (l!!i)
        else max' (tail l) res
```

Version 4

```
max l = max' l (l!!0)

max' []     res = res
max' (x:xs) res =
    if x > res
        then max' xs (l!!i)
        else max' xs res
```

Quelques exemples de fonctions récursives

```
-- longueur d'une liste
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)

-- somme des éléments d'une liste d'entiers
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + (sum xs)
```

Quelques exemples de fonctions récursives

```
-- une liste d'entiers contient-elle la valeur 0 ?
hasZero :: [Int] -> Bool
hasZero []     = False
hasZero (0:xs) = True
hasZero (x:xs) = hasZero xs

-- combien de fois la valeur 0 apparait-elle ?
countZeros :: [Int] -> Int
countZeros []     = 0
countZeros (0:xs) = 1 + (countZeros xs)
countZeros (x:xs) = (countZeros xs)
```

Quelques exemples sur les listes

- On a une liste d'entiers
- On veut la liste avec les valeurs opposées de ces valeurs

  opposés [2, 1, 2, 3, -1] == [-2, -1, -2, -3, 1]

```
opposés :: [Int] -> [Int]
opposés l = opposés' l []

opposés' :: [Int] -> [Int] -> [Int]
opposés' [] res     = res
opposés' (x:xs) res = opposés' xs ((opposé x):res)
```

Quelques exemples sur les listes

- Problème : le résultat est à l'envers !
- C'est dû à l'opération de construction de liste chaînée
- Il faut inverser la liste finale avant de la renvoyer

```
opposés :: [Int] -> [Int]
opposés l = opposés' l []

opposés' :: [Int] -> [Int] -> [Int]
opposés' [] res     = reverse res
opposés' (x:xs) res = opposés' xs ((opposé x):res)
```

Quelques exemples sur les listes

- On a une liste de nombres
- On veut la liste avec les valeurs inverses de ces valeurs

  inverses [2, 1, 2, 3, -1] == [0.5, 1.0, 0.5, 0.33333, -1.0]

```
inverses :: [Double] -> [Double]
inverses l = inverses' l []

inverses' :: [Double] -> [Double] -> [Double]
inverses' [] res     = reverse res
inverses' (x:xs) res = inverses' xs ((inverse x):res)
```

Quelques exemples sur les listes

- On a une liste de jours
- On veut la liste indiquant si ces jours font partie ou non du weekend

  weekends [Lundi, Dimanche, Lundi, Jeudi] == [False, True, False, False]

```
weekends :: [Jour] -> [Bool]
weekends l = weekends' l []

weekends' :: [Jour] -> [Bool] -> [Bool]
weekends' [] res     = reverse res
weekends' (x:xs) res = weekends' xs ((weekend x):res)
```

Tiens, tiens...

- Toutes ces fonctions ont beaucoup en commun
- On a une liste en entrée, une autre liste de même taille en sortie
- Les types d’entrée et de sortie ne sont pas forcément les mêmes
- On applique une fonction à chaque élément

Quelques exemples sur les listes

- On a une liste de a
- On veut une liste de b en appliquant une fonction de transformation f :: a -> b

```
appliquer :: [a] -> [b]
appliquer l = appliquer' l []

appliquer' :: [a] -> [b] -> [b]
appliquer' [] res     = reverse res
appliquer' (x:xs) res = appliquer' xs ((f x):res)
```

Quelques exemples sur les listes

- Problème : comment indiquer quelle est la fonction f à utiliser ?
- Si seulement on pouvait passer la fonction à utiliser en paramètre…
  - « remplacer f par opposé »
  - « remplacer f par inverse »
  - « remplacer f par weekend »
- On peut ! En Haskell, une fonction est un paramètre comme un autre
- On peut donc passer une fonction en paramètre à une autre fonction
- Une fonction qui prend une autre fonction en paramètre est appelée *fonction d'ordre supérieur*
- Ça n'est pas aussi compliqué que ça en a l'air !

Quelques exemples sur les listes

- Plutôt que de récrire trois fois la même chose, utilisons cette fonction générique !
- On veut juste lui passer en paramètre f :: a -> b

```
appliquer :: (a -> b) -> [a] -> [b]
appliquer f l = appliquer' f l []

appliquer' :: (a -> b) -> [a] -> [b] -> [b]
appliquer' f [] res     = reverse res
appliquer' f (x:xs) res = appliquer' xs ((f x):res)
```

Quelques exemples sur les listes

- Maintenant, on peut utiliser appliquer comme on veut
- Il suffit de lui passer le bon paramètre pour f
- On n’a plus besoin d’écrire les fonctions opposés, inverses et weekends !

```
appliquer opposé [1, 2, 3] == [-1, -2, -3]
appliquer inverse [1, 2, 3] == [1, 0.5, 0.333333]
appliquer weekend [Lundi, Samedi] == [False, True]
```

La fonction map

- La fonction qu’on a écrite (« appliquer ») existe déjà
- Elle est fondamentale en programmation fonctionnelle
- Beaucoup de programmes l’utilisent sous une forme ou une autre

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

- On applique une fonction sur plusieurs éléments du même type

La fonction map

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

map opposé [1, -2, 2, 0]
> [-1, 2, -2, 0]

map inverse [1, 2, 5]
> [1, 0.5, 0.2]

map weekend [Lundi, Samedi, Mercredi]
> [False, True, False]
```

La fonction filter

- Il existe d’autres fonctions d’ordre supérieur fondamentales
- Comment éliminer des éléments d’une liste selon un critère donné ?

```
filter :: (a -> Bool) -> [a] -> [a]
```

- On élimine tous les éléments qui ne vérifient pas le test

La fonction filter

```
estPositif :: Int -> Bool
estPositif n = n > 0

-- rappel : filter :: (a -> Bool) -> [a] -> [a]

filter estPositif [4, 0, -1, 2, -2]
> [4, 2]

filter weekend [Lundi, Samedi, Mercredi]
> [Samedi]
```

La fonction filter

- C’est pénible de devoir écrire la fonction avant !
- Si seulement on pouvait mettre la fonction directement dans l’appel à filter (ou à map)…
- On peut !
- On va utiliser une fonction anonyme : une « lambda »

La fonction filter

```
estPositif :: Int -> Bool
estPositif n = n > 0

filter estPositif [4, 0, -1, 2, -2]
> [4, 2]

filter (\n -> n > 0) [4, 0, -1, 2, -2]
> [4, 2]
```

Les lambdas

- Une lambda, c’est une fonction qui n’a pas de nom et qu’on utilise directement comme paramètre d’une fonction

  `\x -> x + 1` est la fonction anonyme qui à x associe x + 1

- Syntaxe : \ param1 -> \ param2 -> … -> expression
- On peut lire le \ comme voulant dire « la fonction qui… »

La fonction filter

```
filter (\n -> n > 0) [4, 0, -1, 2, -2]
> [4, 2]
```

~~`filter (n -> n > 0) [4, 0, -1, 2, -2]`~~

```
-- incorrect : il manque le \
```

La fonction map

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

map (\x -> (- x)) [1, -2, 2, 0]
> [-1, 2, -2, 0]

map (\x -> (1 / x)) [1, 2, 5]
> [1, 0.5, 0.2]

map weekend [Lundi, Samedi, Mercredi]
> [False, True, False]
```

La fonction zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith (\x -> \y -> x + y)) [1, -2, 0, 2] [-1, 3, 2, 1]
> [0, 1, 2, 3]

zipWith (\x -> \y -> x > y) [1, -2, 0, 2] [-1, 3, 2, 1]
> [True, False, False, True]

max :: Int -> Int -> Int
max x y = if x > y
   then x
   else y

zipWith max [1, -2, 0, 2] [-1, 3, 2, 1]
> [1, 3, 2, 2]
```

La fonction foldr

- Il existe d'autres fonctions d'ordre supérieur fondamentales
- Comment réduire toutes les valeurs d'une liste à une valeur synthétique ?
  - Faire la somme de toutes les valeurs
  - Y a-t-il une valeur positive dans la liste ?
  - Y a-t-il un jour de weekend dans la liste ?
  - Combien y a-t-il de feux rouges dans la liste ?

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

La fonction foldr

- Le premier paramètre est la fonction à appliquer sur chaque élément
  - Deux paramètres : l'élément courant et la valeur finale courante
- Le deuxième paramètre est la valeur initiale
  - Celle qui sera renvoyée si la liste est vide
- Le dernier paramètre est la liste

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

La fonction foldr

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

-- somme des valeurs
-- ici a et b valent Int
foldr (\x -> \y -> x + y) 0 [1, 2, 3]
> 6

foldr (\x -> \y -> x + y) 0 []
> 0
```

La fonction foldr

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


-- y a-t-il une valeur positive ?
-- ici a vaut Int et b vaut Bool
foldr (\x -> \y -> (x > 0) || y) False [-5, 4, 0]
> True
```

La fonction foldl

- Parfois, on veut synthétiser en partant de la gauche
- Dans ce cas, on utilise foldl

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (b -> a -> b) -> b -> [a] -> b
```

La fonction foldl

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (b -> a -> b) -> b -> [a] -> b

foldl (\x -> \y -> x - y) 0 [1, 2, 3]
> -6

foldr (\x -> \y -> x - y) 0 [1, 2, 3]
> 2

foldr ((\x -> \y -> y - x) 0 [1, 2, 3]
> -6
```

Fonctions d’ordre supérieur

- Beaucoup de programme de traitement de liste peuvent être écrits en utilisant uniquement ou presque map, filter, foldr et foldl
- Il est très important d’en comprendre le fonctionnement !
- map et filter se parallélisent très facilement
- foldr et foldl sont séquentielles et se parallélisent moins bien

Fonctions d’ordre supérieur

- Quelle est la moyenne des valeurs entre 0 et 20 de la liste ?

```
inBounds :: [Double] -> Bool
inBounds l = filter (\x -> x >= 0 && x <= 20) l

sum :: [Double] -> Int
sum l = foldr (\x -> \y -> x+y) 0 l

moyInBounds :: [Double] -> Double
moyInBounds l = (sum (inBounds l)) / (length (inBounds l))
```

Types avancés (typeclasses)

- J’ai menti

  Le type de (+) n’est pas `Int -> Int -> Int`

- C’est

```
(+) :: Num a => a -> a -> a
```

- + prend un type a tel que a soit numérique (Num a)
- Le type de (==) est

```
(==) :: Eq a => a -> a -> Bool
```

- == nécessite un type sur lequel on peut tester l’égalité et l’inégalité

Types avancés (typeclasses)

- Un type fait partie d’une classe de types s’il implémente certaines fonctions
- C’est une fonctionnalité avancée (pour nous) mais il faut savoir que ça existe !

Types avancés : autres exemples

- Quel est le type de (<) ?

  (<) :: Ord a => a -> a -> Bool
- Quel est le type de (/) ?

  (/) :: Fractional a => a -> a -> a

Types somme

- Rappel sur les énumérations :

```
data Feu = Vert | Orange | Rouge
```

- On peut créer une « énumération » en associant à certains noms une valeur d’un type de notre choix

Carte à jouer

- Une carte, c’est une couleur et un rang
- Une couleur est soit pique, soit cœur, soit carreau, soit trèfle

```
data Couleur = Pique | Coeur | Carreau | Trèfle
```

- Un rang est un roi, une dame, un valet, un As ou un Dix

```
data Rang = Roi | Dame | Valet | As | Dix
```

Carte à jouer

- Une carte, c’est soit un rang et une couleur, soit un joker

```
data Carte = Normale Rang Couleur | Joker
```

- On peut créer une liste de cartes ensuite :

```
[Normale Roi Pique, Normale Dix Trefle, Joker]
```

Carte à jouer

- Quelle est la plus forte entre deux cartes ?

```
plusForte :: Carte -> Carte -> Carte
plusForte Joker _                = Joker
plusForte _ Joker                = Joker
plusForte (Normale Roi x) _ = Normale Roi x
plusForte _ (Normale Roi x) = Normale Roi x
...
```

Nombre complexe

- Un nombre complexe peut être représenté comme une paire (partie réelle, partie imaginaire) ou comme une paire (rho, theta)

```
data Complex = Cartesian Double Double | Polar Double Double
```

- On peut créer une fonction qui convertit de cartésien à polaire et une autre qui fait l’inverse

Nombre complexe

```
toPolar :: Complex -> Complex
toPolar (Cartesian re im) =
   let rho = sqrt (re*re + im*im) in
      let theta = atan (im / re) in
         Polar rho theta
toPolar x = x
```

Nombre complexe

```
toCartesian :: Complex -> Complex
toCartesian (Polar rho theta) =
   Cartesian (rho * (cos theta)) (rho * (sin theta))
toCartesian x = x
```

Figures géométriques

- En Haskell, pas de listes hétérogènes
- Problème : comment représenter une liste de figures géométriques ?
  - Des carrés
  - Des cercles
  - Des triangles

En utilisant des types somme !

```
data Coord  = Point Double Double
data Figure = Carré Coord Double  |
              Cercle Coord Double |
              Triangle Coord Coord Coord


[Carré (Point 1 2) 1,
 Triangle (Point 0 0) (Point 1 2) (Point -1 -3),
 Cercle (Point 0 0) 3]
-- type : [Figure]
```

Un joueur c’est soit un humain, soit l’IA

```
data Joueur = Humain String | IA


let highScores = [
(Joueur "Toto", 17352),
(IA, 17151),
(Joueur "Titi", 14121),
(Joueur "Tutu", 12630),
(Joueur "Tata", 11236)]
-- type : [(Joueur, Int)]
```

Maybe…

- Une fonction ne va pas toujours renvoyer un résultat
  - Quel est le maximum d’une liste vide ?
  - Quelle est la moyenne d’un étudiant qui n’a pas de note ?
  - Quel est l’inverse de 0 ?
- Plusieurs solutions (imparfaites) selon les langages
  - Renvoyer -1
  - Lever une exception

Maybe…

- Une fonction ne va pas toujours renvoyer un résultat
  - Quel est le maximum d'une liste vide ?
  - Quelle est la moyenne d'un étudiant qui n'a pas de note ?
  - Quel est l'inverse de 0 ?
- La solution de Haskell : renvoyer « peut-être » un résultat
  - Le maximum d'une liste est « peut-être » un entier
  - La moyenne d'un étudiant est « peut-être » un double
  - L'inverse d'un entier est « peut-être » un double

Maybe...

```
-- Utilisation d'une valeur sentinelle

moyenne :: [Double] -> Double
moyenne [] = -1.0
moyenne l = (sum l) / (length l)

-- quelle est la moyenne de [-2, -1, 0] ?
```

Maybe...

```
data Maybe a = Just a | Nothing

moyenne :: [Double] -> Maybe Double
moyenne [] = Nothing
moyenne l = Just ((sum l) / (length l))
```

Fonctions d’ordre supérieur

- Un étudiant a subi plusieurs partiels
- Pour chacun, soit il a une note, soit il était absent
- La liste des notes est donc de type `[Maybe Int]`
- Quelle est la moyenne des notes où il était présent ?
- Quelle est la moyenne en mettant zéro quand il était absent ?

Fonctions d’ordre supérieur

```
estPrésent :: Maybe Int -> Bool
estPrésent Nothing  = False
estPrésent (Just _) = True

noteOuZero :: Maybe Int -> Int
noteOuZero Nothing  = 0
noteOuZero (Just x) = x
```

Fonctions d’ordre supérieur

- Quelle est la moyenne des notes où il était présent ?

```
moyennePrésent :: [Maybe Int] -> Double
moyennePrésent notes =
    let notes' = map noteOuZero (filter estPrésent notes) in
        (sum notes') / (length notes')
```

Fonctions d’ordre supérieur

- Quelle est la moyenne en mettant zéro quand il était absent ?

```
moyenneZéros :: [Maybe Int] -> Double
moyenneZéros notes =
    let notes' = map noteOuZero notes in
        (sum notes') / (length notes')
```

Fonctions d’ordre supérieur : zip et unzip

- Souvent, on doit manipuler des listes complexes
  - Pas des listes d’entiers, mais…
  - Des listes de joueurs (nom, score, vies) (String, Int, Int)
  - Des listes d’étudiants (nom, notes) (String, [Double])
  - Des listes de comptes bancaires (nom, balance) (String, Double)
  - Etc.
- Des listes de tuples

Fonctions d’ordre supérieur : zip et unzip

- Comment manipuler facilement ces liste complexes ?
- J’ai une liste d’étudiants, je veux la moyenne de la promo

```
promo = [("Toto", [12, 13, 8]),
         ("Titi", [8, 6, 13]),
         ("Tata", [11, 10, 0])]
```

Fonctions d’ordre supérieur : zip et unzip

- Si j’avais une liste de notes, (une liste de liste de Double) ça serait facile :

```
notes = [[12, 13, 8], [8, 6, 13], [11, 10, 0]]


map moyenne notes
> [11.0, 9.0, 7.0]
moyenne (map moyenne notes)
> 9.0
```

Fonctions d’ordre supérieur : zip et unzip

- Comment séparer les noms des notes ?
- Avec la fonction unzip

```
unzip :: [(a, b]) -> ([a], [b])
promo = [("Toto", [12, 13, 8]), ("Titi", [8, 6, 13]), ("Tata", [11, 10, 0])]
(noms, notes) = unzip promo

let (noms, notes) = unzip promo in
    moyenne (map moyenne notes)
> 9.0
```

Fonctions d’ordre supérieur : zip et unzip

- On veut calculer la liste des moyennes (mais pas faire la moyenne globale)

```
zip ::  [a] -> [b] -> [(a, b)]


let (noms, notes) = unzip promo in

    map moyenne notes

> [11.0, 9.0, 7.0]
```

Fonctions d’ordre supérieur : zip et unzip

- On veut calculer la liste des moyennes (mais pas faire la moyenne globale)
- Comment associer à nouveau chaque moyenne à son nom ?

```
zip ::  [a] -> [b] -> [(a, b)]


let (noms, notes) = unzip promo in
    zip noms (map moyenne notes)
> [("Toto", 11.0), ("Titi", 9.0), ("Tata", 7.0)]
```

Fonctions d’ordre supérieur : zip et unzip

- Tous les étudiants n’ont pas le même nombre de notes
- Certains n’en ont pas : ils n’ont pas de moyenne
- La fonction maybeMoyenne ne renvoie pas forcément une moyenne
- Elle renvoie peut-être (Maybe) une moyenne

```
maybeMoyenne :: [Double] -> Maybe Double

maybeMoyenne [] = Nothing

maybeMoyenne l  = Just ((sum l) / (length l))
```

Fonctions d’ordre supérieur : zip et unzip

- Calculons la liste des moyennes

```
promo = [("Toto", [10, 12]),
         ("Titi", []),
         ("Tutu", [12, 13, 14])]


let (noms, notes) = unzip promo in
   map maybeMoyenne notes
> [Just 11.0, Nothing, Just 13.0]
```

Fonctions d’ordre supérieur : zip et unzip

- Calculons la liste des moyennes et associons au nom

```
promo = [("Toto", [10, 12]),
         ("Titi", []),
         ("Tutu", [12, 13, 14])]

let (noms, notes) = unzip promo in
   zip noms (map maybeMoyenne notes)
> [("Toto", Just 11.0),
   ("Titi", Nothing)
   ("Tutu", Just 13.0)]
```

Fonctions d’ordre supérieur : zip et unzip

- Comment calculer la moyenne de la promo ?
- On ne veut pas de ces Nothing qui traînent au milieu
- On ne veut que les Just x, où x est la valeur qui nous intéresse

```
catMaybes :: [Maybe a] -> [a]

let (noms, notes) = unzip promo in
   map maybeMoyenne notes
> [Just 11.0, Nothing, Just 13.0]

catMaybes (let (noms, notes) = unzip promo in
   map maybeMoyenne notes)
> [11.0, 13.0]
```

Fonctions d’ordre supérieur : zip et unzip

```
moyenne (catMaybes (let (noms, notes) = unzip promo in
    map maybeMoyenne notes))
> 12.0
```

Fonctions d’ordre supérieur : zip et unzip

```
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
unzip3 :: [(a, b, c)] -> ([a], [b], [c])
```

```
zip4, zip5, …, zip7
unzip4, unzip5, …, unzip7
```

Listes en compréhension

- On sait créer des listes en extension :

```
[1, 2, 3, 4, 5, 6, 7]
[2, 3, 4, 5]
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

- Il existe une syntaxe plus légère :

```
[1..7]
[2..5]
[4..15]
```

Listes en compréhension

- On sait créer des listes en extension :

  [1, 2, 3, 4, 5, 6, 7]

  [1, 3, 5, 7, 9]

  [2, 3, 5, 7, 11, 13, 17]

- Comment écrire ces listes de manière plus compacte ?
- En utilisant des listes en compréhension / en intention
  - Quelle est la liste des nombres impairs entre 1 et 10 ?
  - Quelle est la liste des nombres premiers inférieurs à 20 ?

Listes en compréhension

- L’ensemble des x, pour tout x compris dans [1;10], tels que x modulo 2 vaut 1

```
[x | x <- [1..10], mod x 2 == 1]
```

- L’ensemble des x, pour tout x compris dans [2..20], tels que x est premier

```
[x | x <- [2..20], estPremier x]
```

- Les inverses de tous les nombres entre 1 et 100 : L’ensemble des valeurs 1/x pour tout x compris dans [1..100]

```
[1/x | x <- [1..100]]
```

Listes en compréhension

```
promo = [("Toto", [10, 12]),
         ("Titi", []),
         ("Tutu", [12, 13, 14])]
```

- La liste des moyennes des étudiants ayant eu au moins une note ?
- Liste des noms des étudiants :

```
[x | (x, y) <- promo]
["Toto", "Titi", "Tutu"]
```

- Liste des noms des étudiants ayant eu au moins une note :

```
[x | (x, y) <- promo, length y > 0]
["Toto", "Tutu"]
```

Listes en compréhension

```
promo = [("Toto", [10, 12]),
         ("Titi", []),
         ("Tutu", [12, 13, 14])]
```

- Liste des noms des étudiants et de leur moyenne, pour les étudiants ayant eu au moins une note :

```
[(x, moyenne y) | (x, y) <- promo, length y > 0]
> [("Toto", 11.0), ("Tutu", 13.0)]
```

- Liste des moyennes, pour les étudiants ayant eu au moins une note :

```
[moyenne y | (x, y) <- promo, length y > 0]
> [11.0, 13.]
```

Quelle est la moyenne de la promo ?

```
promo = [("Toto", [10, 12]), ("Titi", []), ("Tutu", [12, 13, 14])]
```

- Approche récursive :

```
moyennePromo :: [(String, [Double])] -> Double
moyennePromo l = (sumPromo l) / (lengthPromo l)

sumPromo :: [(String, [Double])] -> Double
sumPromo [] = 0
sumPromo ((nom, []):xs) = sumPromo xs
sumPromo ((nom,notes):xs) = (moyenne notes) + (sumPromo xs)

lengthPromo :: [(String, [Double])] -> Double
lengthPromo [] = 0
lengthPromo ((nom,[]):xs) = lengthPromo xs
lengthPromo ((nom, notes):xs =  1+ (lengthPromo xs)
```

Quelle est la moyenne de la promo ?

```
promo = [("Toto", [10, 12]), ("Titi", []), ("Tutu", [12, 13, 14])]
```

- Approche utilisant des fonctions d'ordre supérieur :

```
moyennePromo :: [(String, [Double])] -> Double
moyennePromo l = let (noms, notes) = unzip l in
    moyenne (map moyenne (filter (\x -> (length x) > 0) notes)
```

- Approche utilisant des listes en compréhension :

```
moyennePromo :: [(String, [Double])] -> Double
moyennePromo l = moyenne [moyenne notes | (noms,notes) <- l, (length notes) > 0]
```

Fonctions d’ordre supérieur

- Quelle est la différence entre ces deux expressions ?

```
map opposé [1, 3, 1, 2, -2]
map (\x -> opposé x) [1, 3, 1, 2, -2]
```

- C’est la même chose !

Fonctions d’ordre supérieur

```
map opposé [1, 3, 1, 2, -2]
map (\x -> opposé x) [1, 3, 1, 2, -2]

opposé :: Int -> Int
(\x -> opposé x) :: Int -> Int
```

Currification

- Découvert par Haskell Curry (tiens donc)
- On va faire de l'application partielle de fonction
- On ne va lui donner qu'une partie de ses arguments d'entrée
- On va obtenir une fonction partielle

```
plus :: Int -> Int -> Int
plus a b = a + b
```

- La fonction plus est une fonction qui, à deux entiers, associe un entier

Currification

- Découvert par Haskell Curry (tiens donc)

```
plus :: Int -> Int -> Int
plus a b = a + b
```

- La fonction plus est une fonction qui, à deux entiers, associe un entier
- **La fonction plus est une fonction qui, à un entier, associe une fonction qui, à un entier, associe un entier**

Currification

- On veut créer une fonction ajouter3 qui, à un entier, associe cet entier augmenté de 3 :

```
ajouter3 :: Int -> Int
ajouter3 x = 3 + x
```

Currification

- On peut aussi procéder par currification en réutilisant plus
- On fait de l'application partielle de fonction, en ne lui donnant que son premier paramètre

```
plus :: Int -> Int -> Int
(plus 3 4) :: Int
(plus 3) :: Int -> Int

ajouter3 :: Int -> Int
ajouter3 = (plus 3)
```

Currification et fonctions d’ordre supérieur

- On a une liste de notes que les étudiants ont obtenues à un examen
- On veut augmenter toutes les notes de 3 points

```
map (\x -> plus 3 x) [12, 7, 14, 8, 3]
```

```
map (plus 3) [12, 7, 14, 8, 3]
```

Quel est le type de l'opérateur (+) ?

- Admettons que sa signature soit :

```
(+) :: Int -> Int -> Int
```

- On peut faire de la currification avec cet opérateur aussi :

```
(3 +) :: Int -> Int
```

```
map (\x -> plus 3 x) [12, 7, 14, 8, 3]

map (plus 3) [12, 7, 14, 8, 3]

map (3+) [12, 7, 14, 8, 3]
```

Autre exemple : max

- Certains étudiants ont eu des notes négatives !
- On veut que toutes les notes négatives soient ramenées à 0

```
[5, 12, -3, 8, 10] => [5, 12, 0, 8, 20]
```

- On veut, pour chaque élément, le max entre 0 et cet élément

```
map (\x -> max 0 x) [5, 12, -3, 8, 10]

map (max 0) [5, 12, -3, 8, 10]
```

Autre exemple avec filter

- Ne garder que les éléments positifs de l
- Ne garder que les éléments > 0 de l

```
filter (\x -> x > 0) l
filter (> 0) l
```

- Si `filter (> 0) l` est une liste d'entiers, alors qu'est-ce que `filter (> 0)` ?
- C'est une fonction !

```
filter (> 0) :: [Int] -> [Int]
```

- `filter (>0)` est la fonction qui, à une liste, associe la même liste sans les valeurs négatives ou nulles

```
oterNegatifsOuNuls :: [Int] -> Int
oterNegatifsOuNuls = filter (> 0)
```

- C'est une autre manière (plus compacte) d'écrire :

```
oterNegatifsOuNuls :: [Int] -> Int
oterNegatifsOuNuls l = filter (\x -> x > 0) l
```

Listes infinies

- Rappel : on peut créer une liste qui contient tous les nombres de 1 à 10 à l'aide de la syntaxe suivante :

```
[1..10]
```

- On peut également créer des listes en compréhension :

```
[x * 2 | x <- [1..10]] -- 10 premiers nombres pairs

[x | x <- [1..10], (mod x 2) == 0] -- nombres pairs entre 1 et 10
```

Listes infinies

- Imaginons que je veuille la somme des valeurs de ces listes :

```
sum [1..10]
> 55
sum [x * 2 | x <- [1..10]]
> 110
sum [x | x <- [1..10], (mod x 2) == 0]
> 30

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

Listes infinies

- Imaginons que je veuille, dans mon programme principal, afficher ces listes :

```
main :: IO ()
main = do
   print [1..10]
   print [x * 2 | x <- [1..10]]
   print [x | x <- [1..10], (mod x 2) == 0]

> ./prog
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
[2, 4, 6, 8, 10]
```

Listes infinies

- Il y a d’autres fonction simples disponibles sur les listes :

```
length [x | x <- [1..10], (mod x 2) == 0]
> 5

head [1..10]
> 1

last [x * 2 | x <- [1..10]]
> 20

take 3 [x * 2 | x <- [1..10]]
> [2, 4, 6]
```

Listes infinies

- Il y a d'autres fonction simples disponibles sur les listes :

```
length [x | x <- [1..10], (mod x 2) == 0]
> 5

length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + (length xs)

head [1..10]
> 1

head :: [a] -> a
head (x:xs) = x
```

Listes infinies

- Il y a d'autres fonction simples disponibles sur les listes :

```
last [x * 2 | x <- [1..10]]
> 20

last :: [a] -> a
last (x:[]) = x
last (x:xs) = last xs

take 3 [x * 2 | x <- [1..10]]
> [2, 4, 6]

take :: Int -> [a] -> [a]
take 0 l = []
take n (x:xs) = x:(take (n-1) xs)
```

Listes infinies

- En Haskell, on peut créer des listes infinies
- `[1..]` est la liste de tous les nombres entiers positifs
- `[x * 2 | x <- [1..]]` est la liste de tous les nombres pairs
- `[x | x <- [1..], estPremier x]` est la liste de tous les nombres premiers
- Mais… si j'écris ça, je vais avoir une boucle infinie ?

Listes infinies

- Mais… si j’écris ça, je vais avoir une boucle infinie ?
- Dans la plupart des langages, oui
- En Haskell… pas forcément !
  - Évaluation paresseuse (vs évaluation stricte)
  - Haskell ne calcule une valeur que quand on l’oblige à le faire
  - Il calcule au dernier moment

Listes infinies

```
-- inutile de calculer toutes les valeurs pour avoir
-- la première !
head [1..]
> 1

-- inutile de calculer toutes les valeurs pour prendre
-- les 5 premières !
take 5 [x | x <- [1..], estPremier x]
> [2, 3, 5, 7, 11]
```

Listes infinies

- Mais… si j’écris ça, je vais avoir une boucle infinie ?
- Dans la plupart des langages, oui
- En Haskell… pas forcément !
  - Évaluation paresseuse
  - Haskell ne calcule une valeur que quand on l’oblige à le faire
  - Il calcule au dernier moment

Listes infinies

- Attention, si on essaie d’aller au bout de la liste, forcément, on aboutira à une boucle infinie !

```
length [1..]

last [x | x <- [1..], estPremier x]

sum [x * 2 | x <- [1..]]

print [1..]
```

Listes infinies

- On peut faire un map sur une liste infinie

```
map (*2) [1..]
```

- On peut faire un filter sur une liste infinie

```
filter (>10) [1..]
```

- On ne peut pas faire un fold sur une liste infinie !
- Il faut parcourir toute la liste pour avoir le résultat

```
foldl (\x -> \y -> x*y) [1..] 1
```

Rappel sur foldl

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

-- produit des nombres entre 1 et 10
foldl (\x -> \y -> x*y) 1 [1..10] 1
> 3628800

-- combien de fois la chaîne contient-elle le caractère ‘ ’ ?
let str = "j’adore Haskell c’est un super langage" in
   foldl (\nb -> \c -> if c == ' ' then nb+1 else nb) 0 str
> 5
```

Listes infinies

- On peut écrire une fonction récursive qui renvoie une liste infinie

```
répèteÀLInfini :: a -> [a]
répèteÀLInfini x = x:(répèteÀLInfini x)


take 4 (répèteÀLInfini "toto")
> ["toto", "toto", "toto", "toto"]
```

Nombres aléatoires

- Parfois on a besoin de nombres aléatoires
  - Jeux incorporant une part de hasard
  - Simulation (physique, météo, etc.)
  - Cryptographie (sans aléa l'algo serait prévisible)
  - Etc.
- Problème : les ordinateurs sont des machines à calculer / à exécuter des algorithmes
- Un algorithme est déterministe

Nombres aléatoires

- Comment écrire un algorithme qui calcule un nombre aléatoire ?
- Un algorithme / une formule mathématique renvoie toujours le même résultat pour des paramètres donnés
- f(x) = y : pour un x donné, y sera toujours le même
- Sommes-nous condamnés au déterminisme ?

Nombres aléatoires

- Il existe deux solutions :
  - Les nombres pseudo-aléatoires : même s’ils ne sont pas aléatoires, ils en ont tout l’air
  - Les nombres aléatoires forts utilisant des dispositifs physiques externes

Nombres pseudo-aléatoires

- Si on n’a pas de nombre aléatoire, il « suffit » d’utiliser une fonction générant une série de nombres en apparence chaotique
- Quel nombre vient après ?

  1, 2, 3, 4, 5, 6, 7, 8, ?

  1, 2, 4, 8, 16, 32, 64, 128, ?

  2, 3, 5, 7, 11, 13, 17, 19, ?

  1, 1, 2, 3, 5, 8, 13, 21, ?

  1, 4, 13, 6, 2, 7, 5, 16, ?

Nombres pseudo-aléatoires

- Quel nombre vient après ?

  1, 4, 13, 6, 2, 7, 5, 16, ?

- Cette suite a l'air chaotique
- Elle ne l'est pas
- $x_i = (3x_{i-1} + 1) \bmod 17$, avec $x_0 = 1$
- Super, on a une fonction pseudo-aléatoire !

Nombres pseudo-aléatoires

- $x_i = (3x_{i-1} + 1) \bmod 17$, avec $x_0 = 1$
- Super, on a une fonction pseudo-aléatoire !
- 1, 4, 13, 6, 2, 5, 16, 15, 12, 3, 10, 14, 9, 11, 0, 1, 4, 13, 6, 2, 5, 16, 15, 12, 3, 10, 14, 9, 11, 0, 1, 4, 13, 6, …
- On a des cycles ! (3*0+1) mod 17 = 1, et 1 est notre point de départ
- Déterminisme : la valeur qui suit 1 est toujours 4
- Pas terrible…

Nombres pseudo-aléatoires

- Changeons nos paramètres
- $x_i = (175341x_{i-1} + 1) \bmod 1000000007$, avec $x_0 = 1$

1,175342,744641413,169082872,159651824,510276034,309451291,498435419,164191108,432866306,210429054,840499143,959201153,288190865,674106249,462978524,116808432,307131946,822166623,516834331,247797518,64299496,337849219,819494014,898902945,540175948,990235271,841437016,406789691,910710350,862361563,337759535,94211875,204258743,932005666,804338181,460007491,172914826,58293434,228939448,471470775,156580600,998792423,261015341,690585920,24951105,951671188,976607047,255029355,101822037,577664647,296160612,97505190,657400119,193458697,241153231,483800784,134987964,924430049,488087080,676095214,210088146,65349928,521645243,797912609,793795328,865632561,877815842,106474704,380943382,993075705,885971524,131902256,873307408,593154247,258095200,670146423,143132716,33380478,966352035,130982849,663565748,281005619,805896183,641634262,792345815,706575406,637396210,688075284,807527316,646123613,959633997,183490137,343886507,403601809,944296501,891622831,37716006,162161756,604259766,510888550,...

Nombres pseudo-aléatoires

- C’est beaucoup mieux !
- Il y a encore des cycles, mais ils sont beaucoup plus grands
- $x_i = (175341x_{i-1} + 1) \bmod 1000000007$, avec $x_0 = 1$
- En Haskell :

```
random :: Int -> Int

random x = mod (175_341*x + 1) 1_000_000_007
```

Nombres pseudo-aléatoires

- C’est pénible de devoir toujours se promener avec la valeur xi !
- Il faudrait créer une liste (infinie) de valeurs aléatoires une bonne fois pour toutes, et « piocher » dedans à chaque fois qu’on a besoin d’un nombre aléatoire

```
randomList :: Int -> [Int]

randomList xi =

   let xi’ = random xi in

      xi’:(randomList xi’)
```

Nombres pseudo-aléatoires

- Cool, on a des valeurs (pseudo-)aléatoires
- Mais ces nombres sont beaucoup trop grands
- Je veux lancer un dé, donc avoir une valeur entre 1 et 6

[1,175342,744641413,169082872,159651824,510276034,309451291,498435419,164191108,432866306,210429054,840499143,959201153,288190865,674106249,462978524,116808432,307131946,822166623,516834331,247797518,64299496,337849219,819494014,898902945,540175948,990235271,841437016,406789691,910710350,862361563,337759535,94211875,204258743,932005666,804338181,460007491,172914826,58293434,228939448,471470775,156580600,998792423,261015341,690585920,24951105,951671188,976607047,255029355,101822037,577664647,296160612,97505190,657400119,193458697,241153231,48380784,134987964,924430049,488087080,676095214,210088146,65349928,521645243,797912609,793795328,865632561,877815842,106474704,380943382,993075705,885971524,131902256,873307408,593154247,258095200,670146423,143132716,33380478,966352035,130982849,663565748,281005619,805896183,641634262,792345815,706575406,637396210,688075284,807527316,646123613,959633997,183490137,343886507,403601809,944296501,891622831,37716006,162161756,604259766,510888550,...]

Nombres pseudo-aléatoires

- Comment s’assurer qu’un nombre est compris entre 1 et 6 ?
- On peut faire un modulo : (x mod 6) + 1 pour x entier est forcément une valeur entre 1 et 6
- Soit randomNums ma liste infinie de valeurs pseudo-aléatoires

```
map (\x -> (mod x 6) + 1) randomNums
```

Nombres pseudo-aléatoires

- C’est largement assez aléatoire pour un jeu !

```
map (\x -> (mod x 6) + 1) randomNums
```

```
[1,5,2,5,3,5,2,6,5,3,1,4,6,6,4,3,1,5,4,2,3,5,2,5,4,5,6,5,6,3,2,6,2,6,5,4,2,
5,3,5,4,5,6,6,3,4,5,2,4,4,2,1,1,4,2,2,1,1,6,5,5,1,5,6,6,3,4,3,1,5,4,5,3,5,2
,5,4,5,1,4,6,3,6,4,5,6,5,5,1,1,6,4,4,6,2,2,2,1,3,1,5,...]
```

Nombres pseudo-aléatoires

- Comment utiliser cette liste dans un programme ?
- À chaque fois que je veux lancer un dé, je prends le premier élément de la liste, et je renvoie le reste de la liste

```
map (\x -> (mod x 6) + 1) randomNums
```

```
[1,5,2,5,3,5,2,6,5,3,1,4,6,6,4,3,1,5,4,2,3,5,2,5,4,5,6,5,6,3,2,6,2,6,5,4,2,
5,3,5,4,5,6,6,3,4,5,2,4,4,2,1,1,4,2,2,1,1,6,5,5,1,5,6,6,3,4,3,1,5,4,5,3,5,2
,5,4,5,1,4,6,3,6,4,5,6,5,5,1,1,6,4,4,6,2,2,2,1,3,1,5,...]
```

Nombres pseudo-aléatoires

- J’ai un dé (ma liste infinie de valeurs aléatoires comprises entre 1 et 6)
- Je marque un point chaque fois que je fais un 6
- J’ai droit à nbLancers lancers
- Quel est mon score ?

```
superJeu :: [Int] -> Int -> Int
superJeu randomDice nbLancers = superJeu’ randomDice nbLancers 0

superJeu’ [Int] -> Int -> Int
superJeu’ randomDice 0 score = score
superJeu’ (x:xs) n score =
    if x == 6
        then superJeu’ xs (n-1) (score+1)
        else superJeu’ xs (n-1) score
```

Nombres pseudo-aléatoires

- J’ai un dé (ma liste infinie de valeurs aléatoires comprises entre 1 et 6)
- Je marque un point chaque fois que je fais un 6
- J’ai droit à nbLancers lancers
- Quel est mon score ?

```
superJeu :: [Int] -> Int -> Int

superJeu randomDice nbLancers =

    sum [1 | x <- (take nbLancers randomDice), x == 6]
```

Nombres pseudo-aléatoires

- La fonction présentée ici « fonctionne »
- Elle est satisfaisante pour beaucoup d’applications
- Mais pas assez pour d’autres
  - Simulation avec beaucoup de tirages
  - Cryptographie
- Dans ces domaines, on a besoin de nombres cryptographiques forts
- Il faut utiliser des bibliothèques dédiées
- Génération des nombres plus lente