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.

Induction et récursion

Axiomatique de Peano

- Publiée en 1889 (*Arithmetices principia, nova methodo exposita*).
- Définition des entiers naturels de $\mathbb{N}$ en 5 axiomes.
- Part d'un objet de base : *zéro*.
- Utilise un « constructeur » : la fonction *succ* qui, à tout entier *n*, fait correspondre son successeur *succ(n)*.
- À l'aide de cet objet et de cette fonction, Peano définit 5 axiomes qui permettent de décrire l'ensemble des entiers naturels.

Axiomatique de Peano

1. *zéro* est un entier naturel ;
2. Tout entier naturel a un successeur, qui est également un entier naturel ;
3. *zéro* n'est le successeur d'aucun entier naturel ;
4. Deux entiers naturels différents ont des successeurs différents ;
5. *Principe d'induction* : si une propriété P est vraie pour *zéro* (*cas de base*) et que $\forall n, P(n) \Rightarrow P(succ(n))$ (*étape d'induction*), alors P est vraie pour tout entier naturel.

Axiomatique de Peano

- On part d'un objet de base (*zéro*, noté 0)
- On lui adjoint un successeur (noté 1), puis le successeur du successeur (noté 2), etc. On définit ainsi l'ensemble des entiers naturels...
- C'est le premier exemple d'ensemble défini *inductivement*.
- D'autres ensembles (listes, arbres, etc.) se définissent de la même façon. C'est le fondement des structures récursives.

Principe d'induction

- Base de la technique de *preuve par récurrence*.
- Pour prouver qu'une propriété P est vraie pour tous les entiers naturels, il faut montrer que :
  - P est vraie pour *0* (en général, c'est assez simple).
  - Si P est vraie pour n (*hypothèse de récurrence*), alors elle est vraie pour son successeur *succ(n)*.
    Autrement dit, il faut montrer que $P(n) \Rightarrow P(n+1)$.
- Selon ce principe, on peut tenter de définir une fonction f par *induction naturelle* :
  - f est définie pour tout entier naturel n si :
    - On connaît *f(0)* ;
    - On peut exprimer $f(n+1)$ à partir de $f(n)$, $\forall n > 0$.

Remarques

- Ici, on n'a pas utilisé les opérations arithmétiques pour définir les entiers...
- L'addition, par exemple, n'est pas une notion primitive de la théorie des nombres puisqu'on peut la définir par récurrence :

$$\begin{array}{lcl} plus(0,p) & = & p \\ plus(succ(n),p) & = & succ(plus(n,p)) \end{array}$$

- Calcul de $2+2$:

$$\begin{array}{lcl} deux & = & succ(succ(0)) \\ plus(deux,deux) & = & plus(succ(succ(0)),succ(succ(0))) \\ & = & succ(plus(succ(0),succ(succ(0)))) \\ & = & succ(succ(plus(0,succ(succ(0))))) \\ & = & succ(succ(succ(succ(0)))) \end{array}$$

Intermède

Écrire la fonction *plus x y* qui additionne deux entiers x et y uniquement en utilisant *succ* et *pred*.

- Il faut se rappeler du temps où l'on comptait sur ses doigts...
- Si $y = 0$, le résultat est x (cas de base), donc *plus x 0* $= x$
- Si $y > 0$, $x + y$ consiste à ajouter 1 à x y fois...
- Si $y < 0$, $x + y$ consiste à ôter 1 à x y fois

Intermède

- On a donc la condition d'arrêt de la récursivité : $y = 0$.
- Ceci signifie donc que les appels récursifs devront ramener y vers 0... :
  - Si $y > 0$, il faudra décrémenter y (avec *pred y*).
  - Si $y < 0$, il faudra incrémenter y (avec *succ y*).
- L'incrémentation/décrémentation de x et y s'effectuera avec *succ*/*pred*.
- D'où :

```
plus :: Int -> Int -> Int
plus x 0 = x
plus x y
  | y > 0     = plus (succ x) (pred y)  -- Incrémente x y fois
  | otherwise = plus (pred x) (succ y)  -- Décrémente x y fois
```

Définition récursive de la factorielle

On veut donner une définition de la fonction factorielle :

- Par une phrase : ≪ *factorielle associe à tout entier n positif ou nul, un autre entier qui est le produit des entiers de 1 à n* ≫.
    - Correct mais difficilement exploitable par le calcul algébrique...
- Par une formule mathématique : $\mathit{facto}(n) = 1 \times 2 \times 3 \times \ldots \times n$
    - Définition *elliptique* : que signifie ...? (avec beaucoup de mauvaise foi, on pourrait conclure que $\mathit{facto}(2) = 1 \times 2 \times 3 \times 2 = 12$).

Définition récursive de la factorielle

- Une spécification est *récursive* lorsqu'elle définit un objet mathématique (ensemble, relation ou fonction) à l'aide de lui-même.
- En observant différents calculs de factorielles, on constate que chaque étape ne diffère de la précédente que par un seul terme : $\mathit{facto}(4)$, par exemple, est égale à $\mathit{facto}(3) \times 4$.
- D'où : $\forall n \in \mathbb{N}, \mathit{facto}(n) = \mathit{facto}(n-1) \times n$

Définition récursive de la factorielle

- Problème du cas $n = 0$: $facto(0) = facto(-1) \times 0$...
  - -1 ne fait pas partie du domaine de définition (entiers naturels)
  - le résultat est incorrect.
- Il faut donc limiter $facto(n) = facto(n-1) \times n$ aux cas où $n > 0$ et préciser ce qui se passe lorsque $n = 0$ (*cas de base*)...
- D'où :

$$\begin{aligned} facto(0) &= 1 \\ facto(n) &= n \times facto(n-1), \forall n > 0 \end{aligned}$$

Factorielle récursive en Haskell

Première remarque

- Il faut vérifier que le paramètre est positif ou nul, sous peine de provoquer des appels récursifs sans fin...

```
facto :: Int -> Int
facto n
  | n == 0    = 1
  | n > 0     = facto (n - 1) * n
  | otherwise = error("n doit être positif ou nul")
```

Deuxième remarque

- On rappelle que l'utilisation de *error* doit être évitée dans la mesure du possible.
- Pour éviter cela, on peut renvoyer un *Maybe* ou une liste (voir plus loin)

Tours de Hanoï : présentation

On utilise N disques de diamètres différents et 3 piquets A, B, C (Eduard Lucas, 1892) :

- Au départ, tous les disques sont sur le piquet de gauche (A), empilés du plus grand au plus petit. Le but consiste à mettre tous les disques sur le piquet de droite (C) en se servant du piquet du milieu (B) comme stockage temporaire.
- On ne peut déplacer qu'un disque à la fois.
- On ne peut placer un disque que sur un disque de diamètre supérieur.
- N disques $\Rightarrow 2^N - 1$ déplacements (voir Wikipédia).
- Avec 64 disques et une seconde par déplacement, il faudrait donc *213 000* milliards de jours...
- ... soit *584,5* milliards d'années...
- ... soit *43* fois l'âge estimé de l'univers (*13,7* milliards d'années).

Tours de Hanoï : principe

Pour que tous les disques du piquet A soient déplacés sur le piquet C :

- Il faut placer le plus grand disque sur le piquet vide C.
- Pour cela, il faut déplacer les $N-1$ disques du piquet A vers B.
- Puis, il faut déplacer le $N-1^e$ disque de B vers C.
- Pour cela, il faut déplacer les $N-2$ disques de B vers A.
- Etc.
- Jusqu'à ce qu'il ne reste plus que le petit disque à déplacer de A vers C.

Tours de Hanoï : algorithme

L'algorithme est donc le suivant :

- Si la tour n'a qu'un disque, le déplacer. C'est fini...
- Sinon :
  - Déplacer tous les disques sauf celui du bas du piquet de départ vers le piquet temporaire
  - Déplacer le disque du bas (qui est maintenant seul) du piquet de départ vers le piquet d'arrivée
  - Déplacer les disques du piquet temporaire vers le piquet d'arrivée.

Solution en Haskell

On suppose que les disques sont numérotés de N (le plus grand) à 1 (le plus petit) :

hanoi.hs

```haskell
type Piquet = String

hanoi :: Int -> Piquet -> Piquet -> Piquet -> IO ()

hanoi 1 debut temp fin =
   putStrLn ("Disque 1 déplacé de " ++ debut ++ " vers " ++ fin)

hanoi n debut temp fin = do
   hanoi (n - 1) debut fin temp  -- déplace les n-1 disques de début vers temp
   putStrLn ("Disque " ++ show n ++ " déplacé de " ++ debut ++ " vers " ++ fin)
   hanoi (n - 1) temp debut fin  -- déplace les n-1 disques de temp vers fin

main = hanoi 3 "A" "B" "C"
```

Solution en Haskell

Exemple d’exécution avec 3 disques

```
$ runhaskell hanoi.hs
Disque 1 déplacé de A vers C
Disque 2 déplacé de A vers B
Disque 1 déplacé de C vers B
Disque 3 déplacé de A vers C
Disque 1 déplacé de B vers A
Disque 2 déplacé de B vers C
Disque 1 déplacé de A vers C
```

Solution en Python et en Go

hanoi.py

```python
def hanoi(n, debut, temp, fin):
    if n == 1:
        print(f"Disque 1 déplacé de {debut} vers {fin}")
    else:
        hanoi(n - 1, debut, fin, temp)
        print(f"Disque {n} déplacé de {debut} vers {fin}")
        hanoi(n -1, temp, debut, fin)

hanoi(3, "A", "B", "C")
```

hanoi.go

```go
package main

import "fmt"

func hanoi(n int, debut, temp, fin string) {
  if n == 1 {
    fmt.Printf("Disque 1 déplacé de %s vers %s\n", debut, fin)
  } else {
    hanoi(n-1, debut, fin, temp)
    fmt.Printf("Disque %d déplacé de %s vers %s\n", n, debut, fin)
    hanoi(n-1, temp, debut, fin)
  }
}

func main() {
  hanoi(3, "A", "B", "C")
}
```

where et let

- *where* permet de créer des liaisons *locales* à une fonction (noms locaux ou fonctions locales).
- *let* permet aussi de créer des liaisons locales, mais ce sont également des expressions. Elles peuvent notamment apparaître dans une liste en intension (voir plus loin).

where et let

```haskell
cylindre, cylindre' :: Double -> Double -> Double

cylindre r h = surfaceCote + (2 * surfaceBase)
  where
    surfaceCote = 2 * pi * r * h
    surfaceBase = pi * r ^ 2

cylindre' r h =
  let surfaceCote = 2 * pi * r * h
      surfaceBase = pi * r ^ 2
  in  surfaceCote + (2 * surfaceBase)
```

Le type Maybe

- Le type *Maybe* permet de représenter les *valeurs facultatives*.
- Une valeur de type *Maybe a* (*a* étant un type quelconque) peut être soit *Nothing*, soit une valeur *Just a*.
- On peut extraire la valeur encapsulée dans le *Just* par un appel à la fonction *Data.Maybe.fromMaybe* (mais c'est rarement nécessaire).

Utilisation de Maybe

```
facto :: Int -> Maybe Int
facto n
 | n < 0     = Nothing
 | otherwise = Just (product [2..n])

ghci> facto 4
Just 24
ghci> facto (-2)
Nothing
```

Curryfication

- En Haskell, toute fonction ne prend en fait qu'*un seul paramètre*.
- Toutes les fonctions à plusieurs paramètres sont des fonctions « curryfiées » : `add x y = x + y` peut en réalité être vue comme une fonction qui n'attend qu'*un seul* paramètre `x` et qui renvoie une fonction d'un seul paramètre auquel `y` est appliqué.
- Du fait de l'associativité à droite de l'opérateur `->`, la signature `Int -> Int -> Int` (deux paramètres `Int`, un résultat `Int`), peut s'écrire `Int -> (Int -> Int)` : un paramètre `Int`, un résultat qui est une fonction d'un paramètre `Int` renvoyant un `Int`.
- `add 5 2` réalise donc l'appel `add 5`, qui renvoie la fonction $(\lambda y \rightarrow 5 + y)$ à laquelle est ensuite appliqué le paramètre 2 pour produire 7.
- Ce mécanisme est généralisable à un nombre quelconque d'éléments.
- En réalité, $add\ x + y = x + y$ peut s'écrire $add = \lambda x \rightarrow (\lambda y \rightarrow x + y)$.

Application partielle d'une fonction

La curryfication permet de créer des *applications partielles* de fonctions :

Application partielle de fonction

```
ghci> let add x y = x + y   Il faut utiliser let pour définir  une fonction depuis ghci...
ghci> :t add
add :: Num a => a -> a -> a
ghci> let inc = add 1      inc = (\y -> 1 + y)
ghci> :t inc
inc :: Integer -> Integer
ghci> inc 41
42
```

Sections

Haskell fournit un raccourci pour représenter une application partielle d'une fonction : on place l'opérateur entre parenthèses et on fournit son opérande gauche ou droite.

Applications partielles avec les sections

```
ghci> (+1) 41
42
ghci> (add 1) 41
42
ghci> (1-) 41      Mais (-1) 41 n'est pas possible. Pourquoi ?
-40
ghci> :t map
map :: (a -> b) -> [a] -> [b]
ghci> map (\x -> x ^ 2) [1..5]
[1,4,9,16,25]
ghci> map (^2) [1..5]
[1,4,9,16,25]
```

Applications partielles avec les sections

```
ghci> :t elem
elem :: Eq a => a -> [a] -> Bool
ghci> elem 'e' "aeiouy"  (ou : elem 'e' ['a', 'e', 'i', 'o', 'u', 'y'] )
True
ghci> 'e' `elem` "aeiouy"
True
ghci> :t (`elem` "aeiouy")
(`elem` "aeiouy") :: Char -> Bool
ghci> :t all
all :: (a -> Bool) -> [a] -> Bool
ghci> all (`elem` "aeiouy") "Haskell"
False
ghci> let isVoyelle = (`elem` "aeiouy")
ghci> all isVoyelle "Haskell"
False
ghci> all isVoyelle "aeio"
True
```

Composition de fonctions

La composition de fonctions $f(g(x))$ qui se note $(f \circ g)(x)$ en mathématiques se note *(f.g) x* en Haskell (ou, plus généralement $f.g \ \$ \ x$) :

Composition de fonctions

```
carre, double, doubleCarre :: Int -> Int

carre x  = x * x
double x = x +  x

doubleCarre x = double.carre $ x
```

Remarque

En vertu de la curryfication, on écrira plutôt `doubleCarre = double.carre` (on supprime les derniers paramètres qui sont identiques à gauche et à droite de la définition).

Fonctionnelles

Supposons que nous ayons écrit une fonction calculant la *somme* des *n* premiers entiers :

Somme des n premiers entiers

```
somme :: Int -> Int
somme n
  | n <= 0    = 0
  | otherwise = n + somme (n - 1)
```

On veut maintenant écrire une fonction *sommeCarres*. Il faut tout récrire :

Somme des carrés des n premiers entiers

```
sommeCarres :: Int -> Int
sommeCarres n
  | n <= 0    = 0
  | otherwise = (n ^ 2) + sommeCarres (n - 1)
```

Si l'on a besoin d'une fonction *sommeCubes*, il faudra encore recommencer en remplaçant *(n ^ 2)* par *(n ^ 3)* et si l'on veut calculer la somme des doubles des *n* premiers entiers, il faudra recommencer en utilisant *(n * 2)*... C'est lassant !

Fonctionnelles

- Les *fonctionnelles* ou (*fonctions d'ordre supérieur* par opposition aux *fonctions de premier ordre* classiques) permettent de résoudre ce problème de duplication du code.
- Une fonctionnelle est une fonction dont *au moins un paramètre* est une fonction ou qui *renvoie* une fonction, ou les deux.
- Ici, on veut effectuer le calcul $f(1) + f(2) + \ldots + f(n)$, avec f étant la fonction *identité*, *carré*, *cube* ou *double*.
- Il suffit donc d'écrire une fonctionnelle *sommeFonc* prenant cette fonction en paramètre :

Fonctionnelle de sommation

```
sommeFonc :: (Int -> Int) -> Int -> Int
sommeFonc f n
  | n <= 0    = 0
  | otherwise = f n + sommeFonc f (n - 1)
```

Fonctionnelles

À partir de cette fonctionnelle, nous pouvons ensuite créer des versions spécialisées :

Sommes diverses

```
somme, sommeCarres, sommeCubes, sommeDoubles :: Int -> Int

somme       = sommeFonc id      -- id est la fonction identité
sommeCarres  = sommeFonc (^2)    -- où sommeFonc (\x -> x ^ 2)
sommeCubes   = sommeFonc (^3)    -- où sommeFonc (\x -> x ^ 3)
sommeDoubles = sommeFonc (*2)    -- où sommeFonc (\x -> x * 2)
```

Remarques

- Ces définitions sont des *applications partielles* de *sommeFonc* : on ne précise pas le dernier paramètre (n).
- La forme `\x -> x ^ 2` est une *lambda-expression* (une fonction anonyme). S'il y a plusieurs paramètres, il faut les séparer par une virgule.

Récursivité terminale

- La récursivité et l'utilisation de « variables » non mutables ont un prix :
  - Pertes de performances à cause des nombreux appels de fonctions.
  - Obligation d'allouer de la mémoire pour stocker les différentes valeurs intermédiaires..
  - Risque de dépassement de pile.
- La *mémoïzation* (voir plus loin) et la *récursivité terminale* permettent généralement de résoudre en partie ces problèmes.