# Table des matières

# 1. Functional Programming

La programmation fonctionnelle est un paradigme de programmation qui traite le calcul comme l'évaluation de fonctions mathématiques et évite les états et les données mutables. Elle met l'accent sur l'utilisation de fonctions pures, qui produisent toujours le même résultat pour les mêmes entrées et n'ont pas d'effets secondaires.

# 2. Expressions

## 2.1. Booleans expressions

Les expressions booléeenes sont définies de la manière suivante:

- `True` et `False` -> constantes
- `not b` -> négation
- `b1 && b2` -> conjonction
- `b1 || b2` -> disjonction

## 2.2. Comparisons

Les comparaisons sont définies de la manière suivante:

- `e1 == e2` -> égalité
- `e1 != e2` -> inégalité
- `e1 < e2` -> inférieur
- `e1 <= e2` -> inférieur ou égal
- `e1 > e2` -> supérieur
- `e1 >= e2` -> supérieur ou égal
- `e1 /= e2` -> appartenance

## 2.3. let expressions

Une expression `let` permet de définir des variables locales ou fonctions et de retourner une valeur basée sur ces définitions.

```
cylinder r h =
  let
    side = 2 * pi * r * h
    base = pi * r^2
  in side + 2 * base
```

# 3. Functions

Les fonctions en haskell sont définies en utilisant la syntaxe suivante:

```
functionName param1 param2 = expression
```

nous pouvons en déduire que les fonctions en haskell doivent:

- Avoir au moins un paramètre
- Retourner une valeur
- Si appelée avec les mêmes arguments, doit toujours retourner la même valeur (fonction pure)

## 3.1. Modèle de substitution

Le modèle de substitution est une méthode pour évaluer des expressions en remplaçant les variables par leurs valeurs correspondantes. Il est particulièrement utile pour comprendre comment les fonctions sont évaluées en programmation fonctionnelle.

**Exemple de substitution:**

```
square x = x * x
sumOfSquares x y = square x + square y
```

Chaque ligne correspond à une étape d'evaluation de l'expression `sumOfSquares 3 (2+2)`

```
sumOfSquares 3 (2+2)
sumOfSquares 3 4
square 3 + square 4
3 * 3 + square 4
9 + square 4
9 + 4 * 4
9 + 16
25
```

## 3.2. Guardes

Les gardes sont une manière de définir des fonctions en utilisant des conditions. Elles permettent de choisir entre plusieurs cas en fonction des valeurs des paramètres.

```
abs n | n >= 0 = n
      | otherwise = -n
```

Cette fonction `abs` retourne la valeur absolue de `n`. Si `n` est supérieur ou égal à 0, elle retourne `n`, sinon elle retourne `-n`.

> ⚠ – **Warning**
> Si les gardes ne couvrent pas tous les cas possibles, une erreur d'exécution peut se produire. Cela peut-être un comportement souhaité dans certains cas pour signaler des erreurs.

## 3.3. Where

Les clauses `where` permettent de définir des variables locales ou des fonctions à la fin d'une définition de fonction. Elles sont utiles pour améliorer la lisibilité du code en évitant la répétition d'expressions.

```
cylinder r h =
  side + 2 * base
where
  side = 2 * pi * r * h
  base = pi * r^2
```

## 3.4. Small exercice

```
f n | mod n 2 == 0 = n - 2
    | otherwise = 3 * n + 1

f n = if even n then n -2
      else 3 * n + 1
```

# 4. Lists

Les listes en Haskell sont des collections ordonnées d'éléments du même type. Elles sont définies en utilisant des crochets `[]` et les éléments sont séparés par des virgules.

```
fruits = ["apples", "oranges", "bananas"]
numbers = [1, 2, 3, 4]
diagonal = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
empty = []
```

L'opérateur `:` est associatif à droite, donc `1 : 2 : 3 : []` est équivalent à `1 : (2 : (3 : []))`.

> ⚠ – **Warning**
> Les listes sont:
> - **Immutables**: Une fois créées, leurs éléments ne peuvent pas être modifiés.
> - **Recursive*s**: Une liste est soit vide, soit composée d'un élément (la tête) et d'une autre liste (la queue).
> - **Homogènes**: Tous les éléments d'une liste doivent être du même type.

## 4.1. List construction

L'opérateur de construction de liste `:` (cons) est utilisé pour ajouter un élément au début d'une liste. Il prend un élément et une liste, et retourne une nouvelle liste avec l'élément ajouté en tête.

```
fruits = "apples" : ("oranges" : ("bananas" : []))
numbers = 1 : (2 : (3 : (4 : [])))
empty = []
```

## 4.2. Standard list operators

Haskell fournit plusieurs opérateurs standard pour manipuler les listes:

- `length xs` : Retourne la longueur de la liste `xs`.
- `last xs` : Retourne le dernier élément de la liste `xs`.
- `init xs` : Retourne la liste `xs` sans son dernier élément.
- `head xs` : Retourne le premier élément de la liste `xs`.
- `tail xs` : Retourne la liste `xs` sans son premier élément.
- `take n xs` : Retourne les `n` premiers éléments de la liste `xs`.
- `drop n xs` : Retourne la liste `xs` sans ses `n` premiers éléments.
- `xs \\ ys` : Retourne la liste `xs` sans les éléments de la liste `ys`.
- `xs !! n` : Retourne le `n`-ième élément de la liste `xs` (indexé à partir de 0).
- `xs ++ ys` : Concatène les listes `xs` et `ys`.
- `elem x xs` : Retourne `True` si l'élément `x` est dans la liste `xs`, sinon `False`.
- `reverse xs` : Retourne la liste `xs` dans l'ordre inverse.

# 5. Tuples

Les tuples en Haskell sont des collections ordonnées d'éléments qui peuvent être de types différents. Ils sont définis en utilisant des parenthèses `()` et les éléments sont séparés par des virgules.

```
fruits = (”apples”, ”oranges”, ”bananas”)
numbers = (1, 2, 3, 4)
diagonal = ((1, 0, 0), (0, 1, 0), (0, 0, 1))
empty = ()
```

> 📝 – **Info**
>
> Les tuples d'un seul élément n'existent pas.
>
> Les tuples offrent une manière de regrouper des valeurs hétérogènes, contrairement aux listes qui sont homogènes.

## 5.1. Tuples n = 2

Dans le cas ou le tuple contient exactement deux éléments, on parle de paires. Haskell fournit des fonctions spécifiques pour manipuler les paires:

- `fst (x, y)` : Retourne le premier élément de la paire `(x, y)`.
- `snd (x, y)` : Retourne le deuxième élément de la paire `(x, y)`.

## 5.2. Exercice

Écrire une fonction `splitAt` qui divise une liste en deux parties à un index donné.

```
splitAt n xs
  | n <= 0 = ([], xs)
  | null xs = ([], xs)
  | otherwise = ((head xs : a), b)
  where (a, b) = splitAt (n-1) (tail xs)
```

# 6. List comprehensions

Les compréhensions de liste en Haskell sont une manière concise de créer des listes en utilisant une syntaxe similaire à celle des ensembles en mathématiques. Elles permettent de générer des listes en spécifiant des règles pour les éléments à inclure.

```
az = ['a'..'z']
squares = [x^2 | x <- [1..10]]
evens = [x | x <- [1..10], even x]
```

Si nous souhaitons créer des listes d'éléments ayant une valeur d'écart différente de 1, nous pouvons utiliser la syntaxe suivante:

```
evens = [2,4..20]
odds = [1,3..19]
```

### 6.0.1. Infinite lists

Haskell supporte les listes infinies grâce à son évaluation paresseuse. Cela signifie que les éléments d'une liste ne sont générés que lorsqu'ils sont nécessaires. Nous pouvons définir des listes infinies en utilisant la syntaxe des compréhensions de liste ou des opérateurs de génération.

```
naturals = [0..]  -- Liste infinie des nombres naturels
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)  -- Suite de Fibonacci infinie
firstTenNaturals = take 10 naturals  -- Prend les 10 premiers nombres naturels
firstTenFibs = take 10 fibs  -- Prend les 10 premiers nombres de Fibonacci
```

## 6.1. Multiple generators

Les compréhensions de liste en Haskell peuvent inclure plusieurs générateurs pour créer des listes basées sur des combinaisons d'éléments provenant de différentes listes. Chaque générateur est séparé par une virgule, et les éléments sont combinés de manière cartésienne.

```
pairs = [(x, y) | x <- [1, 2, 3], y <- ['a', 'b', 'c']]
-- Résultat: [(1,'a'), (1,'b'), (1,'c'), (2,'a'), (2,'b'), (2,'c'), (3,'a'), (3,'b'),
(3,'c')]
```

## 6.2. Predicates

Les prédicats dans les compréhensions de liste sont des conditions qui filtrent les éléments générés par les générateurs. Ils sont placés après les générateurs et permettent de sélectionner uniquement les éléments qui satisfont certaines conditions.

```
evens = [x | x <- [1..20], even x]
-- Résultat: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
multiplesOfThree = [x | x <- [1..30], x `mod` 3 == 0]
-- Résultat: [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

## 6.3. Local declarations

Les déclarations locales dans les compréhensions de liste permettent de définir des variables ou des fonctions temporaires qui peuvent être utilisées dans la génération ou le filtrage des éléments. Elles sont définies en utilisant la clause `let`.

```
[y | x <- ['a'..'z'], let y = toUpper x]
-- Résultat: "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

# 7. Pattern Matching

Le pattern matching (ou correspondance de motifs) est une technique utilisée en programmation fonctionnelle pour décomposer des structures de données et extraire leurs composants. En Haskell, le pattern matching est souvent utilisé dans les définitions de fonctions pour traiter différents cas en fonction de la forme des arguments.

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Quand on parle de `Pattern Matching`, on parle principalement d'équations. Dans la cas de fibonnaci, nous avons trois équations. Les deux premières sont des cas triviales (cas de base), tandis que la troisième est un cas récursif.

> 💡 – **Hint**
> Les différentes équations doivent toutes être définies avec les mêmes paramètres (ici `n`) et être de même type de retour (ici `Int`).

## 7.1. Default case

Il est possible de définir un cas par défaut en utilisant la variable `_`, qui peut correspondre à n'importe quelle valeur. Cela est utile pour gérer les cas non spécifiés explicitement.

```
vowel 'a' = True
vowel 'e' = True
vowel 'i' = True
vowel 'o' = True
vowel 'u' = True
vowel 'y' = True
vowel _ = False
```

## 7.2. Pattern Matching avec les listes

Le pattern matching peut également être utilisé pour décomposer des listes en utilisant les opérateurs `:` (cons) et `[]` (liste vide).

```
head (x:_) = x
tail (_:xs) = xs
```

## 7.3. Case expressions

Les expressions `case` permettent de faire du pattern matching de manière plus explicite et flexible.

```
describeList xs = case xs of
  [] -> "The list is empty."
  [x] -> "The list has one element: " ++ show x
  (x:y:_) -> "The list has multiple elements, starting with: " ++ show x ++ " and " ++ show y
```

## 7.4. Non-exhaustive patterns

Lorsque le pattern matching ne couvre pas tous les cas possibles, une erreur d'exécution peut se produire. Il est important de s'assurer que toutes les possibilités sont prises en compte pour éviter des comportements inattendus.

# 8. Recursion

## 8.1. Fonction a recursivité terminale

Une fonction est dite à récursivité terminale si l'appel récursif est la dernière opération effectuée dans la fonction. Cela permet au compilateur d'optimiser l'utilisation de la pile d'appels, évitant ainsi les débordements de pile pour les appels récursifs profonds.

```
sum n = sum n 0
  where sum n total
          | n <= 0 = total
          | otherwise = sum (n-1) (n + total)
```

## 8.2. Fonction a recursivité non terminale

Une fonction est dite à récursivité non terminale si l'appel récursif n'est pas la dernière opération effectuée dans la fonction. Cela signifie que des opérations supplémentaires doivent être effectuées après l'appel récursif, ce qui empêche certaines optimisations.

```
sum n | n <= 0 = 0
      | otherwise = n + sum (n-1)
```