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

5. Tuples

6. List comprehensions

7. Pattern Matching

8. Recursion