Lire la syntaxe Haskell

Vincent Archambault-Bouffard
IFT 2035 - Université de Montréal

Pour obtenir le code source de ce document

- https://github.com/archambaultv/IFT2035-UdeM
- vincent.archambault-bouffard@umontreal.ca

Variables

- Le nom des variables commence par une minuscule
- Une définition se fait avec le signe =

```
foo = 5

baz = “Hello World”

f arg1 = arg1 + 1

f2 arg1 arg2 = arg1 + arg2
```

Types

- Le nom des types commence par une majuscule
- Les annotations de type (::) sont optionnelles
- Le type des fonctions s'écrit `TypeArg1 -> … -> TypeArgN -> TypeRésultat`

```
foo :: Int
foo = 5

bar :: String
bar = “Hello World”

baz = “Hello World”

f :: Int -> Int
f arg1 = arg1 + 1

f2 arg1 = arg1 + 1
```

Déclarations locales

- L'expression `let in` sert à déclarer des variables locales. Les déclarations sont possiblement mutuellement récursives
- L'indentation est importante (comme en Python)

```
foo =
 let x = 1 in
 let a = c
     b = a
     c = 4
 in
   x + a + b + c
```

Fonctions anonymes

- L'expression `\arg -> body` sert à déclarer des fonctions anonymes

```
foo =
 let x = 1 in
 let a = 2
     b = a
     c = 4
 in
   \arg -> arg + x + a + b + c


 bar = \ arg1 arg2 -> arg1 + arg2
```

Priorité des opérations

- L'application de fonction est toujours prioritaire sur l'application des opérateurs

```
foo x y = x + y

x = foo 1 3 + foo 4 5

x2 = (foo 1 3) + (foo 4 5)

b = x == x2 -- True
```

If then else

- Le if a toujours un else en Haskell

```
x = if b then 1 else 1
```

Listes

- Liste vide : []
- Syntaxe pour les listes : [a, b, c]
- Opérateur de concaténation : ++

```
nil = []

nombres = [1, 2, 3, 4]

fruits = [“pomme”] ++ [“banane”]
```

Structure de données (data)

- Le mot clé `data` déclare une nouvelle structure de donnée (type)
- Il faut donner un nom avec une majuscule au type de la structure de donnée (à gauche du =)
- Il peut y avoir plusieurs constructeurs (à droite du =, séparé par |)
- Chaque constructeur indique le type de ses arguments

```
data ListInt = Nil
             | Cons Int ListInt
```

Structure de données (data)

- Le mot clé `data` déclare une nouvelle structure de donnée (type)
- Il faut donner un nom avec une majuscule au type de la structure de donnée (à gauche du =)
- Il peut y avoir plusieurs constructeurs (à droite du =, séparé par |)
- Chaque constructeur indique le type de ses arguments

```
data Boolean = True
             | False
```

Structure de données (data)

- Le mot clé `data` déclare une nouvelle structure de donnée (type)
- Il faut donner un nom avec une majuscule au type de la structure de donnée (à gauche du =)
- Il peut y avoir plusieurs constructeurs (à droite du =, séparé par |)
- Chaque constructeur indique le type de ses arguments

```
data TreeInt = Leaf Int
             | Node TreeInt Int TreeInt
```

Filtrage par motif

- Pour déconstruire une donnée, il faut utiliser le filtrage par motif avec le mot clé `case`

```
case myList of
 Nil -> …
 Cons x xs -> …
```

Filtrage par motif

- Pour déconstruire une donnée, il faut utiliser le filtrage par motif avec le mot clé `case`

```
case myBool of
 True -> …
 False -> …
```

Filtrage par motif

- Pour déconstruire une donnée, il faut utiliser le filtrage par motif avec le mot clé `case`

```
myTree = Leaf 5

myTree2 = Node (Leaf 1) 6 myTree

case myTree of
 Leaf x -> …
 Node tLeft x tRight -> …
```

Filtrage par motif

- La déclaration de fonction peut utiliser du sucre syntaxique pour le filtrage par motif

```
foo :: ListNil -> Int
foo Nil = 0
foo (Cons x xs) = x

-- Équivalent à
foo :: ListNil -> Int
foo arg =
 case arg of
  Nil -> 0
  Cons x xs -> x
```

Filtrage par motif

- La déclaration de fonction peut utiliser du sucre syntaxique pour le filtrage par motif

```
foo :: Boolean -> Int
foo True = 0
foo False = 1

-- Équivalent à
foo :: Boolean -> Int
foo arg =
 case arg of
  True -> 0
  False -> 1
```

Filtrage par motif

- La déclaration de fonction peut utiliser du sucre syntaxique pour le filtrage par motif

```
foo :: TreeInt -> Int
foo (Leaf x) = x
foo (Node tLeft x tRight) = x

-- Équivalent à
foo :: TreeInt -> Int
foo arg =
 case arg of
  Leaf x -> x
  Node tLeft x tRight -> x
```