Cycle ingénieur - 2ème année

Programmation fonctionnelle

Élements de base du langage Haskell

2023-2024

Plan

Types de base

Types entiers

`Int` : nombres entiers (précision fixe)

- Opérateurs usuels : `+ - * div mod`
- `div` est une division **entière**
- *Nombre fini de valeurs permises*

`Integer` : nombres entiers (précision arbitraire)

- Opérateurs usuels : `+ - * div mod`
- `div` est une division **entière**
- *Nombre infini de valeurs possibles*
- Conversion implicite `Int` → `Integer`
- `Integer` est moins performant que `Int` mais les calculs sont plus sûrs

Types numériques

`Double` : nombres réels (double précision)

- Opérateurs usuels : `+` `-` `*` `/` `**` (exponentiation)
- Conversion implicite `Int` → `Double`
- Pas de conversion `Integer` → `Double`

`Float` : nombres réels (simple précision)

- Utilisation non recommandée car non optimisée

Expression	Résultat	Expression	Résultat
`3 + 5`	`8`	`mod 3 5` (*ou* `3 `mod` 5`)	`3`
`3 - 5`	`-2`	`div 3 5` (*ou* `3 `div` 5`)	`0`
`3 * 5`	`15`	`3.0 + 5`	`8.0`
`3 / 5`	`0.6`		

Autres types de base

`Char` : caractère Unicode

- Délimiteur : `'`
- Conversion explicite `Char` ↔ `Int` :
  - `Data.Char.ord` : conversion `Char` vers `Int`
  - `Data.Char.chr` : conversion `Int` vers `Char`

`bool` : Booléen

- Valeurs : `True` `False`
- Opérateurs logiques : `not` `&&` `||`
  (opérateurs binaires « paresseux » ou « court-circuit »)

Autres types de base

Comparaison sur les types de base

- ***Uniquement entre éléments du même type (à conversion implicite près)***
- Opérateurs : `==` `/=` `<` `<=` `>` `>=`
- Type du résultat : `Bool`
- Ordre sur le type `Char` : ordre naturel sur le résultat de `Data.Char.ord`

Expression	**Résultat**	**Expression**	**Résultat**
`'e'`	`'e'`	`'e' == 101`	`Type mismatch`
`Data.Char.ord 'e'`	`101`	`'e' < 'g'`	`True`
`Data.Char.chr 102`	`'f'`	`'e' < 'G'`	`False`

Expression	**Résultat**
`True  && (div 1 0 == 0)`	`Division by zero`
`False && (div 1 0 == 0)`	`False`
`True  || (div 1 0 == 0)`	`True`
`False || (div 1 0 == 0)`	`Division by zero`

Déclaration de constante

Déclaration globale de constante : =

Les constantes ainsi définies sont des expressions réutilisables.

```
x = 5
y = x * x
```

Expression	Résultat
x	5
y	25
x + y	30

NB: Le type est inféré.

On peut déclarer plusieurs constantes simultanément :

```
(z, t) = (2, 1)
```

Expression	Résultat
z	2
t	1

Déclaration locale de constante : `let ... in`

Injecter des constantes dans une expression qui les contient

- Portée **locale** des déclarations
- Allège l'écriture en permettant des évaluations intermédiaires
- **ATTENTION !** Masque la déclaration globale si elle existe

Le résultat est une expression

- Possibilité d'imbrication des déclarations locales
- Type inféré (comme pour la déclaration globale)

```
x = 5
```

Expression	Résultat	Expression	Résultat
`let x = 2 in x * x`	4	`x`	5
`let y = 2 in y * y`	4	`y`	`Variable not in scope`

```
let x = 2 in let y = x * x in x + y → 6
```

Expression conditionnelle

Expression conditionnelle `if ... then ... else ...`

C'est une expression

- La partie `else` est OBLIGATOIRE.

C'est une expression typée

- La condition doit être de type `Bool`.
- Les expressions alternatives doivent être de même type.

Expression	Résultat
`if 1 > 0 then 1 else -1`	`1`
`if 1 then 1 else -1`	`Condition type is not Bool`
`if 1 > 0 then '1' else -1`	`Type mismatch`

Tuples. Filtrage

Tuples

Type (T_1, T_2, ..., T_n)

- T_1 : type (quelconque) de la première composante
- T_2 : type (quelconque) de la deuxième composante
- etc...

Non-associativité de l'opération

- ((T_1, T_2), T_3), (T_1, T_2, T_3) et (T_1, (T_2, T_3)) sont des types différents.

Type couple (T_1, T_2)

- Fonctions d'accès direct : `fst`, `snd`

Expression	Résultat
`(1, 2.0, '3') == ((1, 2.0), '3')`	`Type mismatch`
`fst (1, 2.0)`	`1`
`snd (1, 2.0)`	`2.0`
`snd (fst ((1, 2.0), '3'))`	`2.0`

Filtrage par déclaration *inversée*

- Déclaration (globale ou locale) des composantes
- Possibilité de non-déclaration avec _
- Imbrication possible
- *Ne permet de filtrer qu'un seul motif à la fois*

```
(x, y, z) = (1, 2.0, '3')
```

Expression	Résultat
x	1
y	2.0
z	'3'

```
((x1, _), z1) = ((1, 2.), '3')
```

Expression	Résultat
x1	1
z1	'3'

Expression	Résultat
`let (d, e, f) = (1, 2, 3) in d + e + f`	6

Filtrage par reconnaissance de motifs

a.k.a. *pattern matching* (≃ `switch` en beaucoup plus puissant)

Exhaustivité

- **L'ensemble des motifs doit couvrir tous les éléments du type.**
- _ permet de traiter les cas restants en une seule fois.

Séquentialité

- Les motifs sont traités **dans l'ordre dans lequel ils sont spécifiés**.
- L'analyse s'arrête **au premier motif qui correspond**.

Unicité

- Un identifiant ne peut apparaître **qu'une seule fois par motif**.
- On introduit une clause booléenne avec | pour expliciter les liens si nécessaire.

Pattern matching : exhaustivité

```
t1 = (1, 2, 3)
t2 = (2, 3, 4)
```

Expression

```
case t1 of
  (0, _, _) -> 0
  (1, _, _) -> 1
```

Résultat

Pattern match(es) are non-exhaustive

```
1
```

```
case t2 of
  (0, _, _) -> 0
  (1, _, _) -> 1
```

Pattern match(es) are non-exhaustive
Non-exhaustive patterns in case

```
case t2 of
  (0, _, _) ->  0
  (1, _, _) ->  1
  _         -> -1
```

```
-1
```

Pattern matching : séquentialité

```
t1 = (1, 2, 3)
```

Expression	Résultat

`case t1 of`
`  (1, _, _) -> 1`
`  (1, 2, _) -> 2`
`  _ -> 0`
	Pattern match is redundant
	1

`case t1 of`
`  (1, 2, _) -> 2`
`  (1, _, _) -> 1`
`  _ -> 0 ;`
	2

Pattern matching : unicité

```
t = (2, 2)
```

Expression	Résultat

`case t of`
`  (x, x) -> x`
`  _ -> 0`
	Conflicting definitions for ‘x’

`case t of`
`  (x, y) | x == y -> x`
`  _ -> 0`
	2