PROGRAMMATION FONCTIONNELLE

via le langage Haskell

OUALI Abdelkader

L2 au département Mathématique-Informatique
UFR des sciences
Université de Caen Normandie

Rappel : syntaxe de base

Liaison locale d'une expression à un nom ("définir les variables locales")

- `let expression; expression; in expression` :

```haskell
f = let y = 1+2
        z = 4+6
    in  y+z
```

- `where` :

```haskell
f = y+z
    where y = 1+2
          z = 4+6
```

- `let ... in ...` : est une expression, donc peut-être utilisée comme l'expression
- `where` utilisé dans une construction syntaxique

Types polymorphes

- Une expression est polymorphe quand elle peut servir sans modifications de définition dans de contextes différents
- Du grec, « poly » = plusieurs et « morphê » = formes.
- **Intérêt :** écrire une seule fonction qui prend en argument des valeurs appartenant de plusieurs types
- Rendre une fonction plus générale pour une application sur différentes données
- **Une variable de type** identifie un type quelconque (ici **a**)
- Un type est **polymorphe** comporte une variable de type

```haskell
f :: a -> a
g :: a -> b
```

Récursivité

- Les langages de programmation **purement fonctionnels n'ont pas d'instruction de répétition**

  ➠ Pas de **for** ni de **while**

- **La seule manière** d'implémenter une répétition est la **récursivité**

  ➠ ou principe de récurrence en math

- De manière informelle, une définition $(p.ex.\ D)$ est **récursive** quand elle utilise le **nom** (D) qu'elle est en train de définir

> Définition "**descendant**" d'un individu : est l'un de ses enfants ou un **descendant** de l'un de ses enfants.

Récursivité : exemple formel

- La définition de la fonction somme :

$$somme : \mathbb{N} \rightarrow \mathbb{N}$$

$$x \rightarrowtail \begin{cases} 0, & \text{Si } x = 0 \\ x + somme(n-1), & \text{Si } x > 0 \end{cases}$$

Récursivité : exemple formel

- La définition de la fonction somme :

$$somme : \mathbb{N} \rightarrow \mathbb{N}$$

$$x \rightarrowtail \begin{cases} 0, & \text{Si } x = 0 \\ x + somme(n-1), & \text{Si } x > 0 \end{cases}$$

- **Forme générale**

$$f : \mathbb{D} \rightarrow \mathbb{C}$$

$$x \rightarrowtail \begin{cases} e_0, & \text{Si } x \in D^0 \\ F(f(e_1), f(e_2), \ldots, f(e_k)), & \text{Si } x \in D' \end{cases}$$

  - e_i est une expression qui dépend uniquement de x
  - $F(f(e_1), f(e_2), \ldots, f(e_k))$ dépend uniquement de x et des $f(e_i)$
  - $D_0 \cup D' = D$ et $D^0 \cap D' = \varnothing$

Récursivité : exemple formel

- La définition de la fonction somme :

$$somme : \mathbb{N} \rightarrow \mathbb{N}$$

$$x \rightarrowtail \begin{cases} 0, & \text{Si } x = 0 \\ x + somme(n-1), & \text{Si } x > 0 \end{cases}$$

- **Forme générale**

$$f : \mathbb{D} \rightarrow \mathbb{C}$$

$$x \rightarrowtail \begin{cases} e_0, & \text{Si } x \in D^0 \\ F(f(e_1), f(e_2), \ldots, f(e_k)), & \text{Si } x \in D' \end{cases}$$

  - $f = somme$
  - $D = C = \mathbb{N}$
  - $e_0 = 0$ et $D_0 = 0$
  - $k = 1, e_1 = x - 1, F(f(e_1)) = f(e_1) + x$ et $D' = \mathbb{N}^*$

Récursivité : exemple formel

- Fonction somme récursive utilisant les gardes :

```
somme x

  | x < 0 = error " valeur négative "

  | x == 0 = 0 -- case de base

  | x > 0 = somme (x - 1) + x -- cas récursif (Hérédité)
```

Récursivité : Base et Hérédité

- Fonction somme récursive utilisant les gardes :

```haskell
| x == 0 = 0 -- Base : initialisation de la récursivité

| x > 0 = somme (x - 1) + x -- Hérédité : calcul à partir de paramètres
                            -- plus "petits" effet domino
```

- Deux types d'algorithmes récursifs :
  - algorithmes récursifs qui se terminent

```haskell
ftermine n
  | n == 2 = 2
  | otherwise = 0.5 * ftermine(n-1) + 2
```

Récursivité : Base et Hérédité

- Fonction somme récursive utilisant les gardes :

```
| x == 0 = 0 -- Base : initialisation de la récursivité

| x > 0 = somme (x - 1) + x -- Hérédité : calcul à partir de paramètres
                            -- plus "petits" effet domino
```

- Deux types d'algorithmes récursifs :
  - algorithmes récursifs qui se terminent

```
ftermine n
  | n == 2 = 2
  | otherwise = 0.5 * ftermine(n-1) + 2
```

  - algorithmes récursifs qui ne se terminent pas

```
fterminepas n
  | n == 2 = 0
  | otherwise = 0.5 * fterminepas(n-3) + 2
```

Récursion terminale

- Prenons la fonction suivante qui donne le plus petit diviseur d'un nombre n plus grand ou égal à un autre nombre d

```haskell
d `divise` n = (mod n d) == 0

petitDiviseurPlusGrandQue n d
  | d `divise` n = d
  | otherwise    = petitDiviseurPlusGrandQue n (d+1)
```

- Cette définition récursive possède une propriété intéressante

➠ Récursive terminale

Filtrage par motif ou Pattern matching

- `[]` est le pattern (modèle) de la liste vide
- `(x:xs)` est le pattern (modèle) des listes non vides

Exemples

- `[1,3,2]` vs `(x:xs)` $\longrightarrow$ `x=1`, `xs=[3,2]`
  car `[1,3,2]=1:[3,2]`
- `[3]` vs `(x:xs)` $\longrightarrow$ `x=3`, `xs=[]`
  car `[3]=3:[]`

Pattern matching

- `[]` est le pattern (modèle) de la liste vide
- `(x:xs)` est le pattern (modèle) des listes non vides

Exemples

- `[1,3,2]` vs `(x:y:xs)` $\longrightarrow$ `x`=1, `y`=3, `xs=[2]`
  car `[1,3,2]=1:3:[2]`
- `[1,1,2]` vs `(x:y:xs)` $\longrightarrow$ `x`=1, `y`=1, `xs=[2]`
  car `[1,1,2]=1:1:[2]`
- `[3]` ne satisfait pas le pattern `(x:y:xs)`

Pattern matching

- `[]` est le pattern de la liste vide
- `(x:xs)` est le pattern des listes non vides
- `(x:xs)` est le pattern des listes ayant au moins un élément
- `(_:xs)` le premier n'a pas de liaison, le reste des éléments est lié xs
- `(x:y:xs)` est le pattern des listes ayant au moins deux éléments

- Pour déterminer si une liste possède 2 premiers éléments identiques, peut-on utiliser le pattern `(x:x:xs)` ?
- Quelles listes représente le pattern `[x]` ?
- Quelles listes représente le pattern `[x, y]` ?

Traitement récursif de listes

Somme d'une liste d'entiers

- type de cette fonction : `[Int] -> Int`
- deux définitions équivalentes :

```
sum1 :: [Int] -> Int
sum1 l = if (l == [])
         then 0
         else (head l) + (sum1 (tail l))

-- pattern matching

sum3 :: [Int] -> Int

sum3 [] = 0
sum3 (x:xs) = x + (sum3 xs)
```

Traitement récursif de listes

UNIVERSITÉ CAEN NORMANDIE

Longueur d'une liste

- son type : `[a] -> Int`
- deux définitions équivalentes :

```
lg1 liste = if (liste == [])
            then 0
            else 1 + lg1 (tail liste)
```

```
-- pattern matching
```

```
lg3 [] = 0
lg3 (x:xs) = 1 + (lg3 xs)
```