Type de donnée abstrait et Listes

Programmation Fonctionnelle
Master 2 I2L apprentissage

Sébastien Verel
verel@lisic.univ-littoral.fr
http://www-lisic.univ-littoral.fr/~verel

Université du Littoral Côte d'Opale
Laboratoire LISIC
Equipe OSMOSE

Septembre 2017

Objectifs de la séance

- Savoir différencier une structure de donnée d'un type de donnée abstrait
- Savoir définir le type de donnée abstrait liste
- Savoir utiliser les listes en Haskell
- Connaitre le schéma récursif du traitement d'une liste
- Connaitre les algorithmes classiques relatifs aux listes
- Savoir écrire une fonction de calcul avec accumulateur, de création, de modification, de filtre avec une liste.

Question principale du jour :

La tête ou la queue ?

Exemple de données

- Liste de tâches à effectuer :
  Suite finie de tâches à effectuer dans l'ordre
- Base de données clients
- Génome : suite finie de bases ATCG

Définitions

Donnée

Information élémentaire primitive

Données

Ensemble d'informations élémentaires primitives

Traiter les données consiste à :

- Passer d'informations, appelées données
- à d'autres informations, appelées résultats

Comment accéder (lire, écrire) à ces données ?

Définitions

Donnée

Information élémentaire primitive

Données

Ensemble d'informations élémentaires primitives

Traiter les données consiste à :

- Passer d'informations, appelées données
- à d'autres informations, appelées résultats

Comment accéder (lire, écrire) à ces données ?

Type de Données Abstrait (TDA)

Description des fonctions admissibles sur les données
(et non pas directement sur les structures de données)

Structure de données

Structure de données

Structure logique destinée à contenir les données

Type de Données Abstrait (TDA)

Description des fonctions admissibles sur les données
(et non pas directement sur les structures de données)

Une structure de données implémente un TDA

Fonctions sur les TDA

Opérations élémentaires d'un TDA

- Création de données de base
- Construction de nouvelles données
- Lecture de données

Classification

Les structures logiques contenant les données peuvent être :

- Linéaire / non linéaire
- Indexée / non indexée
- Ordonnée / non ordonnée

Traiter une liste de tâches

Comment traiter une liste de tâches ?

- Faire les tâches les unes après les autres
  traitement séquentiel
- Quelle tache ?
  Pas de préférence (priorité), la première
- Comment ajouter une nouvelle tâche ?
  Pas de préférence, en premier
- Retirer une tâche
  celle qui vient d'être lue
- Savoir s'il reste une tâche

Algorithme du traitement d'une liste de tâche

Algorithme traiter(liste) : rien
début
  si liste est vide **alors**
    Ne rien faire
  sinon
    Exécuter la première tâche
    Traiter le reste de la liste
  fin si
fin

Remarques :

- Algorithme récursifs ?
- Terminaison si la liste est de taille finie (j'espère...)

Définition informelle

Définition

Une liste est une suite finie de données, appelée aussi élément, où il est seulement possible d'ajouter et de lire une donnée en tête de la suite.

Notation : la tête est à gauche et la queue à droite

Exemple

$$[23, 1, 67, 29, 12]$$

La tête est égale à 23
La queue est égale à la liste $[1, 67, 29, 12]$

TDA liste

5 primitives sont nécessaire pour définir le TDA :

- `listeVide` : $() \rightarrow$ *liste*
  retourne la liste vide
- `listeCons` : *element* $\times$ *liste* $\rightarrow$ *liste*
  ajoute un élément à une liste
- `listeTete` : *liste* $\rightarrow$ *element*
  retourne l'élément en tête de la liste (si elle n'est pas vide !)
- `listeQueue` : *liste* $\rightarrow$ *liste*
  retourne la queue de la liste (si elle n'est pas vide !)
- `listeEstVide?` : *liste* $\rightarrow$ *boolean*
  teste si la liste est vide

En Haskell

- `listeVide` : () → *liste*
  `[ ]`
- `listeCons` : *element* × *liste* → *liste*
  `h:t`
- `listeTete` : *liste* → *element*
  `head liste`
- `listeQueue` : *liste* → *liste*
  `tail liste`
- `listeEstVide?` : *liste* → *boolean*
  `liste == []`

Et surtout ne pas oublier le "pattern matching" (filtrage) !

```
h : tl = list
```

Tester !

Algorithme du traitement d'une liste de tâche

Algorithme traiter(liste) : rien
début
  si liste est vide **alors**
    Ne rien faire
  sinon
    Exécuter la première tâche
    Traiter le reste de la liste
  fin si
fin

Algorithme du traitement d'une liste de tâche

Admettons que nous avons une fonction "executer" qui exécute la tache.

```
Algorithme  traiter(liste) : rien
début
   si listeEstVide ?(liste) alors
      executer('fin de tache')
   sinon
      executer( listeTete(liste) )
      traiter( listeQueue(liste) )
   fin si
fin
```

Schéma de traitement récursif des listes

Algorithme traiter(liste) : rien
début
  si listeEstVide ?(liste) **alors**
    ....
  sinon
    ....
    traiter(... listeQueue(liste) ...)
  fin si
fin

En Haskell

```
traiter :: [ String ] -> IO ()
traiter [] = putStrLn "fin des tâches."
traiter (h : t) = do
     executer h
     traiter t
```

En Haskell

```
traiter :: [ String ] -> IO ()
traiter [] = putStrLn "fin des tâches."
traiter (h : t) = do
     executer h
     traiter t

executer tache = putStrLn (tache ++ " [Done]")
```

Longueur d'une liste

Longueur d'une liste

```
longueur :: [ a ] -> Int
longueur [] = 0
longueur (h:t) = longueur t + 1
```

Somme des éléments d'une liste d'entiers

Somme des éléments d'une liste d'entiers

```
mysum :: Num a => [ a ] -> a
mysum [] = 0
mysum (h:t) = mysum t + h
```

Doubler tous les nombres d'une liste d'entiers

Doubler tous les nombres d'une liste d'entiers

```
double :: Num a => [ a ] -> [ a ]
double [] = []
double (h:t) = ( 2*h : double t )
```

Concaténation des mots d'une liste de mots

Concaténation des mots d'une liste de mots

```
concatenationMots :: [ String ] -> String
concatenationMots [] = ""
concatenationMots (h:t) = h ++ concatenationMots t
```

Filtrer une liste : extraire les nombres pairs

Filtrer une liste : extraire les nombres pairs

```
pair :: Integral a => [ a ] -> [ a ]
pair [] = []
pair (h:t) | mod h 2 == 0 = (h : (pair t))
           | otherwise = pair t
```

Concaténation de 2 listes

Concaténation de 2 listes

```
myconcat :: [ a ] -> [ a ] -> [ a ]
myconcat [] l = l
myconcat (h:t) l = ( h : (myconcat t l) )
```

Ajouter un élément à une position donnée

Ajouter un élément à une position donnée

```
ajouter :: a -> Int -> [ a ] -> [ a ]
ajouter elem 0 l = (elem : l)
ajouter elem n (h:t) = (h : (ajouter elem (n-1) t))
```

Supprimer un élément à une position donnée

Supprimer un élément à une position donnée

```
supprimer :: Int -> [ a ] -> [ a ]
supprimer _ [] = []
supprimer 0 (h:t) = t
supprimer n (h:t) = ( h : (supprimer (n-1) t) )
```