Algorithmique Avancée

Nicolas Labroche

Cours 4 : parcours de listes chaînées par itérateurs

Université de Tours

Objectifs du cours

- `Iterator` versus itérateurs
- programmation fonctionnelle et itérateurs
- itérateur d'application : map
- itérateur de sélection : filter
- itérateur de réduction : reduce

Implémentation des `Iterator` dans les listes chaînées

Rappel sur les `Iterator`

Pour rappel, L'interface `Collection` définit plusieurs méthodes et notamment une méthode

```
Iterator<E> iterator()
```

qui permet de générer un *itérateur* qui est un outil de parcours d'une collection.

Rappel sur les `Iterator`

Un `Iterator` pemet de cacher le mécanisme de parcours bas niveau d'une structure et permet d'exprimer l'équivalent mathématique pour une collection `C` de l'expression suivante :

$$\forall c \in C, \text{afficher}(c)$$

ce qui se traduirait en Java par les instructions suivantes

```
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
   System.out.println(iterator.next());
}
```

Quelle différence entre `Iterator` et itérateurs fonctionnels

- Un `Iterator` d'une collection ne doit pas être confondu avec les itérateurs au sens de la programmation fonctionnelle
- Les itérateurs fonctionnels sont des mécanismes de plus haut niveau d'abstraction qui correspondent à des parcours particuliers ayant 3 vocations principales (voir après dans le cours) :
  1. l'application d'une même méthode sur tous les éléments d'une liste pour produire une nouvelle liste (`map`)
  2. le filtrage des éléments d'une liste en fonction d'un prédicat appliqué à chaque élément qui la compose (`filter`)
  3. la réduction ou l'agrégation des éléments d'une liste en une seule valeur (`reduce`)

Implémentation d'un `Iterator`

- Implémenter un `Iterator` nécessite d'ajouter une méthode `Iterator<E> iterator()` sur nos structures de listes chaînées ;
- Il faut donc créer **notre propre classe** qui étend `Iterator` qui va se charger de se déplacer dans notre liste chaînée
- Mais pour qu'une classe externe se déplace dans notre structure de liste d'élément en élément, il faut que cet `Iterator` dispose d'un pointeur vers l'élément actuellement en cours de lecture dans notre liste chaînée

Implémentation de la classe `ISList<T>`

- La classe `ISList<T>` va représenter une liste chaînée implémentant des itérateurs
- Elle étend `SList<T>` mais doit aussi implémenter une interface telle que `Iterable<T>` pour reconnaître les méthodes de Java propres aux itérateurs
- On ajoute donc
  - une interface `IList<T>` qui regroupe les méthodes des listes et des itérateurs
  - une classe `ISList<T>` qui ajoute un champ protected vers le nœud courant
  - une classe `DummyIterator<T>` qui implémente réellement les méthodes de l'itérateur

Interface `IList<T>`

On définit l'interface `IList<T>` comme suit

```
public interface IList<T> extends LList<T>, Iterable<T> {}
```

C'est une interface qui étend deux autres interfaces : notre interface `LList<T>`, et l'interface Java `Iterable<T>` !

Classe `ISList<T>`

On définit la classe `ISList<T>` comme suit

```
public class ISList<T> extends SList<T> implements IList<T> {

    protected Node<T> current;

    public ISList(){ super();}

    public ISList(T value) { super(value, null); }

    public ISList(T value, ISList<T> tail) { super(value, tail↩
        ); }

    @Override
    public Iterator<T> iterator() {
        return new DummyIterator<>(this);
    }
```

Classe `DummyIterator<T>` (1)

```
public class DummyIterator<T> implements Iterator<T> {

    private ISList<T> internal;

    public DummyIterator(ISList<T> internal) {
        this.internal = internal;
        this.internal.current = this.internal.head;
    }

    @Override
    public boolean hasNext(){
        return this.internal.current != null;
    }

[...]
}
```

Classe `DummyIterator<T>` (2)

```
public class DummyIterator<T> implements Iterator<T> {
[...]
    @Override
    public T next() {
      if (this.internal.current != null){
            T tmp = internal.current.value;
            internal.current = internal.current.next;
            return tmp;
      } else return null;
    }
}
```

Fonctionnelles et itérateurs

Petits rappels de programmation fonctionnelle

La programmation **fonctionnelle** est un paradigme de programmation au même titre que

- la programmation **impérative** : description des opérations en séquences,
- la programmation **procédurale** : introduction de sous-routines ou procédures :
- ou la programmation **orientée objets** : décomposition d'un problème en briques logicielles qui modélisent un concept ou toute entité du monde réel

Ces paradigmes de programmation ne sont pas exclusifs, par exemple `Python` et `Scala` sont à la fois objets et fonctionnels.

Propriétés mathématiques des fonctions

La programmation **fonctionnelle** repose sur des fonctions $\mathcal{F}$ au sens mathématique du terme

$$\mathcal{F} : \alpha \to \beta$$

- $\mathcal{F}$ met en relation un élément de l'ensemble (ou de type) α avec un **unique** élément (de type) β
- en revanche un même élément de β peut avoir plusieurs antécédent dans α !
- en mathématiques on parle de *fonction injective*

Propriétés informatiques des fonctions

D'un point de vue programmation, la **fonction** est une opération qui **déclare** une opération

- **indépendante** : car ne dépend d'aucun état d'exécution en dehors du sien,
- **sans état** (ou *stateless* en anglais) : car ne possède aucun état d'exécution interne qui est conservé entre les appels,
- **déterministe** car elle renvoie toujours le même résultat dès lors que l'on fournit les mêmes arguments

Une fonction peut être stockée dans une variable (la fonction pas son évaluation) et passée en argument d'une autre fonction.

Avantages de la programmation fonctionnelle

- code avec minimisation des effets de bords dus à des variables mal initialisées
- applications plus faciles à maintenir et plus robustes dans un contexte parallèle
- plus facile d'optimiser le code par la recherche de fonctions communes

Quels avantages pour le parcours de listes chaînées ?

- possibilité de représenter des parcours classiques par des fonctions **itérateurs**
- ces fonctions de parcours sont paramétrables par une autre fonction qui spécifie l'action à réaliser lors du parcours

Les itérateurs sur les listes chaînées

Cas particulier du schéma général de parcours d'une liste

- schéma d'application : itérateur **map**
  - transformer une liste en une autre par application d'une fonction à chaque élément
- schéma de filtrage : itérateur **filter**
  - sélectionner des éléments d'une liste à l'aide d'un prédicat pour construire une sous-liste
- schéma de réduction : itérateur **reduce**
  - calculer une valeur par applications composées d'une fonction sur les éléments d'une liste

Itérateur map

Contexte

Considérons une liste d'entiers de type `ISList<Integer>` et nous souhaitons pouvoir y appliquer les traitements suivants

- construire une liste qui contient un booléen indiquant si les valeurs de la liste initiale sont positives ou non
- construire une liste qui contient le carré des valeurs de la liste initiale.

Contexte

Par exemple on veut pouvoir écrire

```
CoursIte L = new CoursIte();
L.add(1); L.add(3); L.add(-5); L.add(6); L.add(-2);
System.out.println(L);
```

Et ensuite avoir une méthode qui applique une fonction sur chacun des éléments de la liste initiale pour obtenir `(true, true, false, true, false)` ou `(1, 9, 25, 36, 4)` selon le cas

Liste des éléments positifs / négatifs

- construire une liste qui contient un booléen indiquant si les valeurs de la liste initiale sont positives ou non

```
public ISList<Boolean> listePositifs(){
    Iterator<Integer> it = iterator();
    ISList<Boolean> res = new ISList<>();
    while (it.hasNext()){
        boolean tmp = it.next() >= 0;
        res.add(tmp);
    }
    return res;
}
```

- simple parcours de liste avec l'itérateur (`Iterator`) intégré à notre structure de liste chaînée

Liste des carrés des éléments

- construire une liste qui contient le carré des valeurs de la liste initiale

```
public ISList<Integer> listeCarres(){
    Iterator<Integer> it = iterator();
    ISList<Integer> res = new ISList<>();
    while (it.hasNext()){
        int tmp = it.next().intValue();
        res.add(tmp*tmp);
    }
    return res;
}
```

- mêmes instructions que précédemment, seul le calcul (très simple) change
- peu d'intérêt de reprogrammer les parties identiques du traitement liées au parcours de la liste

Vers un schéma de résolution commun

Les 2 définitions de méthodes précédentes suivent une structure commune qui dépend d'une fonction $\mathcal{M}$

- $\mathcal{M} : x : \mathit{Integer} \rightarrow \mathit{Boolean}\ (x >= 0)$
- $\mathcal{M} : x : \mathit{Integer} \rightarrow \mathit{Integer}\ (x^2)$

Le parcours de la liste est ensuite identique !

Solution

- passer en argument d'une méthode de parcours le nom de la méthode qui fait le traitement $\mathcal{M}$
- possible désormais en `Java` grâce aux interfaces `Function` and `BiFunction`
- autre solution : utilisation d'interfaces et d'objets les implémentant pour représenter les méthodes que l'on souhaite passer en paramètre : beaucoup plus complexe, mais intéressant également intellectuellement !

Implémentation de la méthode `map`

- `Function<T,R> mapper` représente une méthode qui prend en argument un type `T` et produit une valeur de sortie de type `R`

```
public <R> SListFunc<R> map(Function<T,R> mapper){
    Iterator<T> it = this.iterator();
    SListFunc<R> res = new SListFunc<>();
    while (it.hasNext()){
        T val = it.next();
        res.add(mapper.apply(val));
    }
    return res;
}
```

Exemple d'utilisation de `map`

- Utiliser les lambda expressions de Java pour définir les méthodes à utiliser comme mapper

```
/* Map original int list to character list */
Function<Integer, Character> mapper = (num) -> (char)(65+num↩
    %26);
System.out.println(L.map(mapper));
```

Autre solution : définir une interface `Mapper`

Nous proposons d'utiliser une interface Java nommée `Mapper`

```
public interface Mapper<I,O> {
  public O mapTo(I input);
}
```

- une seule méthode déclarée `mapTo()` qui joue le rôle d'une fonction $\mathcal{M} : x : I \rightarrow m(x) : O$

Exemples de mise en place de `Mapper`

Il suffit de construire une classe qui implémente l'interface `Mapper`

- pour la méthode `listePositifs()`, il faut un mapper qui transforme tout entier x en valeur booléenne

```
public class MapPositive implements Mapper<Integer, Boolean> {
    public Boolean mapTo(Integer input) {
        return (input >= 0);
    }
}
```

Exemples de mise en place de `Mapper`

- pour la méthode `listeCarres()`, il faut un mapper qui transforme tout entier *x* en son carré

```
public class MapSquare implements Mapper<Integer, Integer> {
    public Integer mapTo(Integer input) {
        return input * input;
    }
}
```

Le parcours de la liste commun

Intégré à une **méthode générique** de la classe `ISList<T>`

```
public <O> ISList<O> map(Mapper<T,O> mapper){
    Iterator<T> it = iterator();
    ISList<O> res = new ISList<O>();
    while(it.hasNext()){
        T val = it.next();
        res.add(mapper.mapTo(val));
    }
    return res;
}
```

Exemple complet d'utilisation des mappers

```
public static void testMapper(){
    CoursIte L = new CoursIte();
    L.add(1); L.add(3); L.add(-5); L.add(6); L.add(-2);
    System.out.println(L);
    System.out.println(L.listeMapPositifs());
    System.out.println(L.listeMapCarres());
}

private ISList<Integer> listeMapCarres() {
    return this.map(new MapSquare());
}

private ISList<Boolean> listeMapPositifs() {
    return this.map(new MapPositive());
}
```

Itérateur filter

Filtrer une liste selon un prédicat

Nous considérons la définition suivante :

- filtrer une liste : ne conserver que les éléments qui vérifient un prédicat $\mathcal{P}$

Soit une liste *L* d'éléments de type α et soit le prédicat $\mathcal{P} : \alpha \rightarrow$ *Boolean*, la méthode est la suivante

- on applique $\mathcal{P}$ sur chaque élément de la liste
- si le résultat de l'évaluation de $\mathcal{P}$ sur l'élément est vrai (`true`) alors on ajoute cet élément à la liste résultat
- comme précédemment, on sépare le parcours de liste et l'action réalisée lors du parcours

Implémentation de la méthode `map`

- `Function<T, Boolean> filter` représente une méthode qui prend en argument un type `T` et produit une valeur de sortie booléenne

```
public SListFunc<T> filter(Function<T, Boolean> filter){
    Iterator<T> it = this.iterator();
    SListFunc<T> res = new SListFunc<>();
    while (it.hasNext()){
        T val = it.next();
        if (filter.apply(val)){
            res.add(val);
        }
    }
    return res;
}
```

Exemple d'utilisation de `filter`

- Utiliser les lambda expressions de Java pour définir les méthodes à utiliser comme filter

```
/* Filter original list to retain only even values */
Function<Integer, Boolean> filter = (val)-> (val % 2) == 0;
System.out.println(L.filter(filter));
```

Autre solution : définir une interface `Filter`

- Similairement à ce qui est fait pour les `Mapper`, on représente le prédicat $\mathcal{P}$ par une interface Java

```
public interface Filter<I> {
  // return true if the entry input verifies a condition
  // expressed in the filter function
  public boolean filter(I input);
}
```

- Contrairement au `Mapper` il n'y a qu'un seul type générique - en entrée - car la sortie est toujours booléenne

Exemples de prédicats de filtre

Considérons l'exemple d'une liste d'entiers et des prédicats suivants

- $\mathcal{P} : x : \mathit{Integer} \rightarrow \mathit{Boolean} : x\%2 == 0$
  filtre les valeurs paires
- $\mathcal{P} : x : \mathit{Integer} \rightarrow \mathit{Boolean} : x\%2 == 1$
  filtre les valeurs impaires
- $\mathcal{P} : x : \mathit{Integer} \rightarrow \mathit{Boolean} : x >= 0$
  filtre les valeurs positives

Et leurs “interfaces” Java correspondantes

- Filtre des valeurs paires

```
public class FilterEven implements Filter<Integer>{
        public boolean filter(Integer input) {
            return (input % 2) == 0;
        }
}
```

- Filtre des valeurs positives

```
public class FilterPositive implements Filter<Integer>{
        public boolean filter(Integer input) {
            return input >= 0;
        }
}
```

Définition de la méthode de parcours dans `ISList<T>`

```
public ISList<T> filter(Filter<T> filter){
    Iterator<T> it = iterator();
    ISList<T> res = new ISList<T>();
    while(it.hasNext()){
        T val = it.next();
        if (filter.filter(val)) res.add(val);
    }
    return res;
}
```

Exemples de tests de filtres

```
public static void testFilter(){
    CoursIte L = new CoursIte();
    L.add(1); L.add(3); L.add(-5); L.add(6); L.add(-2);
    System.out.println(L);
    System.out.println(L.filterEven());
    System.out.println(L.filterPositifs());
}

public ISList<Integer> filterPositifs(){
    return this.filter(new FilterPositive());
}

public ISList<Integer> filterEven(){
    return this.filter(new FilterEven());
}
```

Itérateur reduce

Motivation

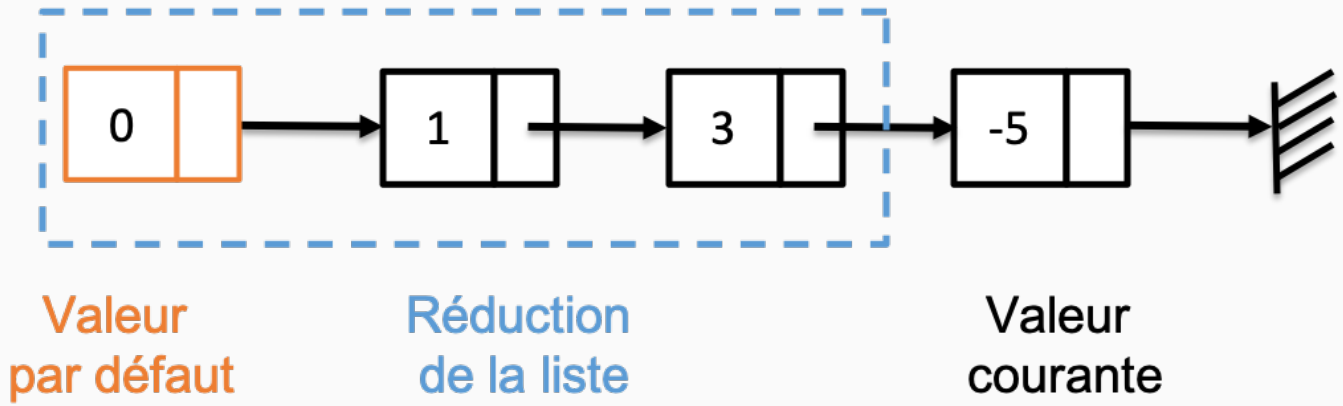
Que faire désormais quand on souhaite calculer une valeur à partir de la liste chaînée ? Par exemple

- trouver la somme des éléments d'une liste d'entiers
- trouver le maximum d'une liste d'entiers

L'idée va consister à **réduire** progressivement les valeurs de la liste en une seule valeur résultat

Exemple introductif

Pour calculer la somme $\mathcal{S}$ des éléments de cette liste, il faut



- avoir une valeur par défaut quand la liste est vide, ici l'élément neutre 0
- réduire la liste progressivement :
  $\sum_{x\in\{1,3,-5\}} x = 0+\sum_{x\in\{1,3,-5\}} x = (0+1)+\sum_{x\in\{3,-5\}} x =$
  $(0+1+3)+\sum_{x\in\{-5\}} x = (0+1+3+-5) = -1$

Implémentation de la méthode `reduce`

- `BiFunction<T, R, R> reducer` représente une méthode qui accepte deux arguments de type et `R` et produit une valeur de sortie de type `R`

```
public <R> R reduce(BiFunction<T, R, R> reducer, R base){
    R res = base;
    Iterator<T> it = this.iterator();
    while (it.hasNext()){
        T val = it.next();
        res = reducer.apply(val, res);
    }
    return res;
}
```

Exemple d'utilisation de `reduce`

- Utiliser les lambda expressions de Java pour définir les méthodes à utiliser comme reducer

```
/* Reduce a list to its size. Base is the value when list is ↩
    empty */
BiFunction<Integer, Integer, Integer> reducer = (val, sum) -> ↩
    sum + 1;
System.out.println(L.reduce(reducer, 0));
```

Autre solution : définir une interface Java

Contraitement aux `Mapper` et `Filter`, le `Reducer` nécessite **deux** méthodes

- une méthode $\mathcal{R} : \alpha \times \beta \to \beta$ qui permet de composer un élément de liste de type α avec la réduction opérée sur les éléments de la liste déjà considérés ou la valeur par défaut de type β.
- cette méthode doit retourner un type β pour qu'à son tour, ce résultat puisse être combinée avec une éventuelle valeur suivante dans la liste
- une méthode $\mathcal{D} :\to \beta$ qui ne prend aucun argument et retourne une valeur par défaut de type β, par exemple l'élément neutre de la somme

Traduction en interface Java

Il faut donc définir une classe qui implémente les méthodes de l'interface suivante

```
public interface Reducer<I,O> {
  public O reduce(I input, O other);
  public O getBase();
}
```

Comme nous le verrons après, il peut être utile que la classe qui implémente cette interface dispose d'autres méthodes, notamment si l'on souhaite fixer la valeur retournée par la méthode `getBase()` à autre chose qu'une valeur constante

Exemple de la somme des éléments d'une liste

- élément neutre de la somme $= 0$
- si on connaît la somme des éléments précédents, déterminer la somme jusqu'à l'élément courant est trivial
  - à l'étape i : $\sum_{k \in [\![1,i]\!]} x_k = \left(\sum_{k \in [\![1,i-1]\!]} x_k\right) + x_i$

```
public class ReducerSum implements Reducer<Integer, Integer> {

    public Integer reduce(Integer input, Integer other) {
        return input + other;
    }

    public Integer getBase() {
        return 0;
    }
}
```

Exemple du maximum des éléments d'une liste

- Problème : la méthode `getBase()` ne peut pas renvoyer une valeur constante car cela pourrait fausser le calcul du maximum (en introduisant un nouveau maximum)
- Solution : ajouter un constructeur qui permet de fixer cette valeur de base
- Lors de l'appel au constructeur, il suffit de fournir la première valeur contenue dans la liste

Exemple du maximum des éléments d'une liste

```
public class ReducerMax implements Reducer<Integer, Integer> {
    private int base;

    public ReducerMax(Integer valBase) { this.base = valBase;}

    public Integer reduce(Integer input, Integer other) {
        return (input >= other)? input:other; }

    public Integer getBase() { return base; }
}
```

Définition de la méthode de parcours dans `ISList<T>`

- la méthode `reduce` permet d'agréger les résultats déjà agrégés aux itérations précédentes avec la nouvelle valeur courante lue dans la liste

```
public <K> K reduce(Reducer<T,K> r){
    K res = r.getBase();
    Iterator<T> it = iterator();
    while (it.hasNext()){
        res = r.reduce(it.next(), res);
    }
    return res;
}
```

Exemples de tests de réduction

```
public static void testReducer(){
    CoursIte L = new CoursIte();
    L.add(1); L.add(3); L.add(-5); L.add(6); L.add(-2);
    System.out.println(L);
    System.out.println(L.reducerSum());
    System.out.println(L.reducerMax());
}

public Integer reducerSum(){
    return this.reduce(new ReducerSum());
}

public Integer reducerMax(){
    if (!isEmpty()) // max not defined on empty list
        return this.reduce(new ReducerMax(this.get(0)));
    else return null;
}
```

Pour conclure

Attention aux signatures des méthodes

Pour bien comprendre les itérateurs et savoir les utiliser, il faut maîtriser les signatures des méthodes qu'ils utilisent

Itérateur	**Entrée**	**Méthode**	**Sortie**
map	*LList*$[\alpha]$	$\mathcal{M} : \alpha \to \beta$	*LList*$[\beta]$
filter	*LList*$[\alpha]$	$\mathcal{P} : \alpha \to$ *Boolean*	*LList*$[\alpha]$
reduce	*LList*$[\alpha]$	$\mathcal{R} : \alpha \times \beta \to \beta$	β