Programmation et modélisation orientées objet
(USAL5A & USAL2H)

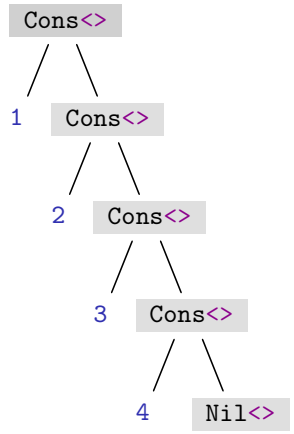
Itérateurs et Visiteurs

1. Des listes *simplement chaînées*[1] (immutables) peuvent être implantées sous la forme de l'interface `SimpleList<E>` et des deux types record `Nil<E>` et `Cons<E>` (cette terminologie vient du langage Lisp). La définition complète est donnée en annexe.

   Par exemple, la liste `[1; 2; 3; 4]` est alors construite ainsi :

   ```
   new Cons<>(1, new Cons<>(2, new Cons<>(3, new Cons<>(4, new Nil<>()))))
   ```

   Cette liste peut aussi être représentée sous forme graphique :



2. Les *records* ont été introduits dans Java 14. Ils peuvent toutefois être traduit en Java 8 ainsi : remplacer `record` par `class` (avec les mêmes attributs « finaux », car les *records* sont immutables) et utiliser la génération de code de l'IDE pour ajouter le constructeur par défaut et les méthodes `equals`, `hashCode` et `toString`.

   Traduire la définition donnée en annexe en source Java 8.

3. Ajouter des méthodes statiques `of` à `SimpleList<E>` permettant de construire facilement des listes immutables de moins de 5 éléments.

4. Tester vos méthodes statiques `of`, en particulier sur l'exemple donné.

5. Ajouter les méthodes suivantes à l'interface `SimpleList<E>` et les implanter dans les classes `Nil<E>` et `Cons<E>` :

   ```
   public boolean isEmpty();
   public SimpleList<E> tl();
   public E hd();
   ```

   Les méthodes `hd` et `tl` doivent renvoyer respectivement la tête (`head`) et le reste (`tail`) de la liste si elle n'est pas vide (donc dans la classe `Cons<E>`) et elles renverront `null` sinon (donc dans la classe `Nil<E>`).

1. La classe `LinkedList<E>` de la JDK correspond à des listes doublement chaînées.

6. Ajouter et implanter la méthode `size` dans l'interface `SimpleList<E>` :

   ```
   public int size();
   ```

   Implanter cette méthode `size` dans les classes `Nil<E>` et `Cons<E>`.

7. (*optionnel*). Optimiser l'implantation de `size` en stockant la taille de la liste.

8. Nous allons maintenant implanter l'interface `Iterable<E>` de la JDK, qui permet en particulier d'utiliser la syntaxe de la boucle `for` « généralisée » (*for-each loop*[2]). La seule méthode à implanter est `iterator`, qui renvoie un itérateur (interface `Iterator<E>` de la JDK).

   Cette interface `Iterator<E>` est rappelée ci-après.

   ```
   public interface Iterator<E> {

      boolean hasNext();
      E next();

   }
   ```

   La classe qui implante cette interface sera appelée `SimpleIterator<E>`, et elle possèdera un attribut `rest` qui contient le reste de la liste qui reste à parcourir.

9. Ajouter et implanter la méthode `iterator` dans l'interface `SimpleList<E>` :

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

   Tester l'itérateur en utilisant la syntaxe de la boucle `for` « généralisée » mais aussi directement avec une boucle `while`.

10. Ajouter la méthode `toString` dans l'interface `SimpleList<E>` :

    ```
    public String toString();
    ```

    Implanter cette méthode dans les classes `Nil<E>` et `Cons<E>` (où on pourra utiliser par exemple la boucle `for` « généralisée »).

11. Généraliser la méthode `list` précédente à un nombre arbitraire d'éléments :

    ```
    static <E> SimpleList<E> of(E... elements)
    ```

12. Implanter les méthodes statiques suivantes dans une nouvelle classe appelée `ListUtil` en utilisant le `switch` avec filtrage (*pattern matching*) et la récursivité :

    - `static <E> Integer length(SimpleList<E> l)`
      renvoie la longueur de la liste `l`.

    - `static <E> SimpleList<E> concat(SimpleList<E> l1, SimpleList<E> l2)`
      renvoie la liste obtenue en concatenant les listes `l1` et `l2`.

    - `static <E> SimpleList<E> rev(SimpleList<E> l)`
      renvoie la liste retournée. La fonction `rev` est sa propre inverse : vérifier cette propriété sur des exemples (en évaluant `rev(rev(l))` pour ces exemples).

2. https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html

- `static <E> SimpleList<E> flatten(SimpleList<SimpleList<E>> l)`
  renvoie la liste obtenue en concatenant tous les éléments de la liste `l`. Autrement dit :
  `flatten([l1;...;ln−1;ln]) = concat(l1, ...concat(ln−1, ln)...)`
- `static <E> E nth(SimpleList<E> l, Integer i)`
  renvoie le `i`-ème élément de la liste `l`. On supposera que $0 \leqslant$ `i` $<$ `length l`.
- `static <E> E last(SimpleList<E> l)`
  renvoie le dernier élément de la liste `l` (que l'on supposera non vide). Donner deux autres implantations de cette fonction, l'une en utilisant `rev`, l'autre en utilisant `nth`.
- `static <E> SimpleList<E> drop(SimpleList<E> l, Integer i)`
  renvoie la liste obtenue en enlevant les `i` premiers éléments de `l`. En particulier, on doit avoir `drop(l, length(l)) = []`. On supposera que $0 \leqslant$ `i` $\leqslant$ `length l`.
- `static <E> SimpleList<E> take(SimpleList<E> l, Integer i)`
  renvoie la liste composée des `i` premiers éléments de `l`. En particulier, on doit avoir `take(l, length(l)) = l`. On supposera que $0 \leqslant$ `i` $\leqslant$ `length(l)`.

13. Ajouter le code nécessaire dans l'interface `SimpleList<E>` du sous-package `java8` pour implanter le patron de conception (*design pattern*) « visiteur ». La méthode `accept` doit implanter le prototype suivant :

    ```
    public <R> R accept(ListVisitor<R, E> v);
    ```

    L'interface d'un visiteur est la suivante :

    ```
    public interface ListVisitor<R, E> {
       R visitNil();
       R visitCons(E h, SimpleList<E> t);
    }
    ```

    Ajouter l'interface `ListVisitor<R, E>` et la méthode `accept` à `SimpleList<E>` puis implanter `accept` dans les classes `Nil<E>` et `Cons<E>`.

    Ré-implanter les méthodes statiques de la question précédente, dans une nouvelle classe `ListUtil` du sous-package `java8`, en utilisant maintenant le patron « visiteur » au lieu du `switch` avec filtrage.

Annexe

```
sealed interface SimpleList<E> {
    record Nil<E>() implements SimpleList<E> {}
    record Cons<E>(E head, SimpleList<E> tail) implements SimpleList<E> {}
}
```