Le principe du flux, l'arbre et la forêt

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/IF/03/

1 Flux de caractères

Un flux est essentiellement caractérisé par une opération `read` qui renvoie « le truc suivant ». En Java les flux de caractères sont réalisés par la classe ***Reader***.

1. Voici la documentation de la méthode `read` d'un objet de la classe ***Reader***.

   public int `read()` **throws** `IOException`

   Read a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.

   Returns : The character read, as an integer in the range 0 to 16383 (0x00-0xffff), or -1 if the end of the stream has been reached

   Throws : IOException If an I/O error occurs (sic).

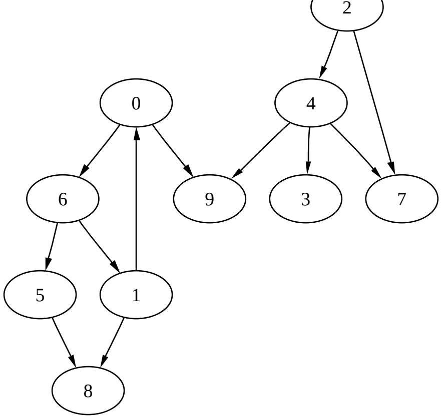
   Qu'est-ce au fond qu'un **char** de Java ? « `-1` » est-il un caractère ? Pourquoi `read` renvoie-t-elle un **int** ?
2. Écrire une méthode `dump` qui prend un flux en argument et affiche son contenu sur la sortie standard.
3. À partir de quoi pensez-vous que l'on peut créer un flux ?

2 Un graphe dans un fichier

On décide de décrire les graphes orientés par des fichiers texte et selon les conventions suivantes :
- Les mots possibles sont les entiers, la flèche `->`, et le point-virgule `;`. Les mots peuvent être séparés par des blancs (espaces, sauts de lignes, etc.)
- Le fichier commence par un entier : le nombre de sommets du graphe.
- Il se poursuit par une suite de définition d'arcs de la forme « s `->` d_1 d_2 ... d_k `;` » définisssant les arcs $s \rightarrow d_1$, $s \rightarrow d_2$, $\ldots s \rightarrow d_k$.

Par exemple :

```
10
0 -> 9  6 ;
1 -> 8 0 ;
2 -> 7  4 ;
4 -> 9 7  3 ;
5 -> 8 ;
6 -> 5 1 ;
```



1. Concevoir une classe ***Token*** des mots (ou lexèmes), écrire le code.

2. Concevoir une classe ***Lexer*** des analyseurs lexicaux réalisée selon le principe du flux. On ne se donnera pas la peine d'écrire le code des méthodes.
3. Donner le code d'un constructeur `Graph (Reader in)` pour la classe suivante des graphes.

```
class Graph { // Graphes orientés
  private List [] succ ;

  Graph (int size) { succ  = new List[size] ;}

  void arc(int src, int dst) {
    succ[src] = new List (dst,succ[src]) ;
  }
```

3 Arbre et forêt

Voici deux classes définissant les arbres n-aires

```
class Tree { // Arbres n–aires
  int node ;
  Forest children ;

  Tree (int node, Forest children) {
    this.node = node ; this.children = children ;
  }
}

class Forest { // Une liste de Tree
  Tree first ;
  Forest next ;

  Forest (Tree first, Forest next) {
    this.first = first ; this.next = next ;
  }
}
```

Une forêt est donc une liste ordonnée d'arbres. Dans un arbre, l'ordre gauche-droit des sous-arbres compte, il compte aussi dans une forêt. Enfin la racine de chaque arbre est identifiée par un entier, dans le champ `node`.

La définition des arbres est inductive, on pourrait aussi l'écrire ainsi :

$$\begin{array}{lcll} T & ::= & (n, F) & n \text{ un entier} \\ F & ::= & \epsilon & \text{forêt vide} \\ & \mid & (T, F) & \end{array}$$

(En supposant en outre que tous les entiers n sont distincts.)

1. Y-a-t-il des arbres, des forêts, vides ? Quelles sont les conséquences pour la programmation en Java ?
2. Définir inductivement l'ordre postfixe des sommets d'une forêt, qui place les descendants *avant* les ancêtres et les sommets des arbres d'une forêt de gauche à droite. On cherchera à définir inductivement une suite ordonnée de sommets, à partir des définitions des arbres et des forêts.
3. Écrire deux méthodes `postOrder`, une pour les forêts l'autre pour les arbres, qui renvoient une liste des sommets dans l'ordre postfixe (à l'aide de notre amie la classe des listes enrichie comme bon nous semble).

4 Forêt profonde

Voici quelques méthodes supplémentaires pour la classe ***Graph***.

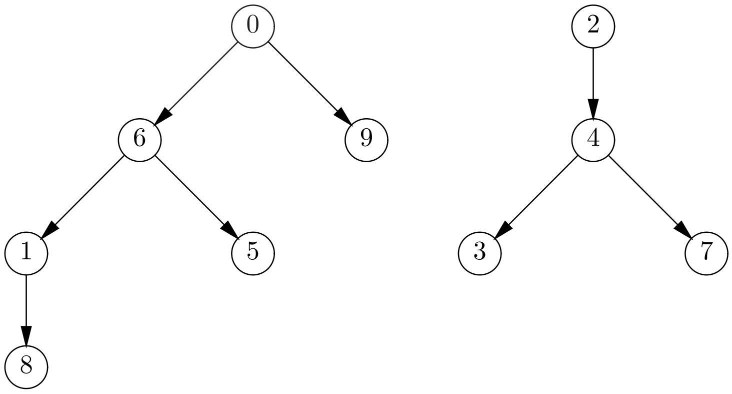
```
Tree dfsTree(int node, boolean [] vu) {
  return new Tree(node, dfsForest(succ[node], vu)) ;
}

Forest dfsForest(List nodes, boolean [] vu) {
  if (nodes == null) return null ;
  int node = nodes.val ;
  if (!vu[node]) {
    vu[node] = true ;
    Tree t = dfsTree(node, vu) ;
    Forest f = dfsForest(nodes.next, vu) ;
    return new Forest (t, f) ;
  } else
    return dfsForest(nodes.next, vu) ;
}

Forest dfs(List nodes) {
  boolean [] vu = new boolean [succ.length] ;
  return dfsForest(nodes, vu) ;
}

Forest dff() {
  List allNodes = null ;
  for (int i = succ.length-1 ; i >= 0 ; i--)
    allNodes = new List (i, allNodes) ;
  return dfs(allNodes) ;
}
```

1. Que réalise ce code ?
2. Voici une forêt produite à partir du graphe de l'exercice 2.



   Les arcs de la forêt ne regroupent pas tous les arcs du graphe d'origine, il manque les arcs `1 -> 0`, `5 -> 8`, `4 -> 9` et `2 -> 7`. Ajouter ces arcs à la forêt et caractériser les nouveaux arcs en fonction de leur nature, arcs de retour, transverses, et de descente.
3. Classifier précisément les arcs du graphe par rapport à la forêt, et énoncer la propriété-clé des arcs transverses.
4. L'ordre préfixe des sommets d'une forêt, place les descendants *après* les ancêtres et les sommets des arbres d'une forêt de gauche à droite. Donner une définition précise.

5. Caractériser les quatres sortes d'arcs produites par `dff` à l'aide des deux ordres préfixe et postfixe sur les sommets. Autrement dit, finir de remplir ce tableau :

$n \to m$	préfixe	postfixe
F	$n \leq m$	$n \geq m$
D		
R		
T		

6. Quel est le rapport entre arcs de retour et les cycles du graphe ?

5 Tri topologique

1. Un tri topologique est une suite n_1, n_2, …, n_k des sommets du graphe, telle que si il existe un chemin de n à m, alors n précède m dans la suite. Quid des cycles ?
2. En examinant le tableau de la question 4-5, réaliser un tri topologique en une ligne.
3. À quoi peut bien servir le tri topologique ?

Solutions des exercices

Exercice 1

1. Les objets de type **char** sont en fait des entiers représentables sur seize bits (de 0 à $2^{16} - 1$). Ces entiers sont en quelque sorte les « codes » des caractères selon une convention mondiale.

   Le « `-1` » que peut renvoyer `read` n'est pas un caractère, il indique au contraire qu'il n'y a plus de caractères.

   La méthode `read` renvoie un entier car c'est un bon moyen de pouvoir renvoyer à la fois les 2^{16} caractères possibles, et « autre chose » dans le cas où le flux est fini.

2. ```
   static void dump(Reader fr) throws IOException { // Hum numéro un
     int c ;

     while ((c = fr.read()) != -1) { // Hum numéro deux
       System.out.print((char)c) ;    // Hum numéro trois
     }
   }
   ```

   Les hums :

   Un Le corps de `dump` peut lever l'exception `IOException` car cette méthode contient un appel à `fr.read`. Il faut déclarer ce fait (noter bien **throws** avec un « s »).

   Deux Le test est pour le moins étrange. Il se comprend quand on sait qu'une affectation renvoie la valeur affectée. Ce style est à éviter sauf dans des tournures idiomatiques du langage de programmation.

   ```
   int c ;

   c = fr.read() ;
   while (c != -1) {
     System.out.print((char)c) ;    // Hum numéro trois
     c = fr.read() ;
   }
   }
   ```

   Un autre emploi courant de la valeur de retour de l'affectation est :

   ```
   x = y = z = 0 ; // À comprendre comme x = (y = (z = 0)) ;
   ```

   Trois La variable `c` contient un **int**. Si on le passe directement à la méthode `print` (de l'objet `System.out`), alors on affiche cet entier. Il faut bien dire que l'entier est ici vu comme un caractère, d'ou la construction `(char)c` que l'on appelle un *type cast* (conversion de type).

3. *A priori* on peut construire un flux de caractères à partir de tout ce qui peut être vu comme une suite de caractères les principaux exemples sont les fichiers (quoique, les fichiers sont plutôt des suites d'octets) et les chaînes. Mais il y a aussi les tableaux de caractères (anecdotique) et les flux d'octets par exemple, combinables selon un *encodage* pour former des caractères.

   On pourrait donc penser qu'il existe divers consructeurs de la classe ***Reader*** prenant en argument une chaîne, un nom de fichier etc. En fait il n'en est rien, la classe ***Reader*** n'a pas de constructeur. Elle sert non pas à créer des objets mais à créer d'autre classes par *héritage*. Des sous-classes sont définies comme héritant de ***Reader*** (et donc, entre autres, comme possédant une méthode `read`).
   ```

Pour ce qui nous concerne, cela revient un peu à considérer une interface ***Reader*** et diverses classes ***FileReader***, ***StringReader*** etc. qui l'implémentent (mais le mécanisme est en fait plus puissant).
À titre d'exemple voici un petit programme qui affiche le contenu d'un fichier dont le nom est donné comme premier argument de la ligne de commande :

```
import java.io.* ;

class Cat {

  static void dump(Reader fr) throws IOException { // Hum numéro un
    int c ;

    while ((c = fr.read()) != -1) { // Hum numéro deux
      System.out.print((char)c) ; // Hum numéro trois
    }
  }

  public static void main(String [] argv) {
    try {
      Reader fr = new FileReader(argv[0]) ;

      dump(fr) ;
      fr.close() ;
    } catch ( IOException e ) {
      System.err.println("Erreur d'entrée sortie : " + e.getMessage()) ;
    }  catch (ArrayIndexOutOfBoundsException e) {
      System.err.println ("Mauvais nombre d'arguments");
    }
  }
}
```

Notez d'abord le **import** `java.io.* ;`. Cette incantation est nécessaire pour accéder à la classe ***Reader***.
Notez que le flux est crée comme un ***FileReader*** et rangé dans une variable de type ***Reader***. C'est possible car, par héritage, un ***FileReader*** est au moins un ***Reader*** (ici, il a les méthodes qui nous intéressent avec les bonnes signatures).
Notez aussi la fermeture de `fr` (par la méthode `close`). Cela permet de dire au système Java de récupérer les ressources mobilisées lors de la création du flux, et les rend libres pour la création de nouveaux flux (ici c'est un peu inutile). Il faut en effet savoir qu'un programme donné (Unix, ici la machine Java) ne peut posséder qu'un nombre limité de fichiers ouverts simultanéement.

Exercice 2

1. C'est pas dur. On fait ici le choix d'un lexème donné pour la fin de flux. Un autre choix est d'exprimer la fin de flux par **null** mais ce sera vite pénible.

```
class Token {
  final static int INT=0, ARROW=1, EOF=2, SEMI=3 ;

  int nature ;
  int asInt ;

  Token (int nat) {
```

```
    nature = nat ;
  }

  Token (int nat, int val) {
    nature = nat ; asInt = val ;
  }
}
```

On note les deux constructeurs distincts à cause des types distincts de leurs arguments (on parle de surcharge).

2. Il faut juste prévoir une méthode `read` et un constructeur,qui range le ***Reader*** dans un champ privé.

```
class Lexer {
  private Reader in ;
  Lexer (Reader in) { this.in = in ; }
  Token read() { ... }
}
```

La version web du cours donne le code des méthodes.

3. Et voilà :

```
  Graph (Reader in) {
    Lexer lexer = new Lexer (in) ;

    succ = new List[readInt(lexer.read())] ;
    Token tok = lexer.read() ;
    while (tok.nature != Token.EOF) {
      int src = readInt(tok) ;
      readArrow(lexer.read()) ;
      tok = lexer.read() ;
      while (tok.nature != Token.SEMI) {
        int dst = readInt(tok) ;
        arc(src,dst) ;
        tok = lexer.read() ;
      }
      tok = lexer.read() ;
    }
  }
```

Où les méthodes auxiliaires `error`, `readInt` et `readToken` sont :

```
  private static void error(String msg) {
    System.err.println("Erreur␣de␣syntaxe␣:␣" + msg) ;
    System.exit(2) ;
  }

  private static int readInt(Token tok) {
    if (tok.nature != Token.INT) error("Entier␣attendu");
    return tok.asInt ;
  }

  private static void readArrow(Token tok) {
    if (tok.nature != Token.ARROW) error("Flèche␣attendue");
  }
```

Exercice 3

1. Selon la définition inductive, il n'y a pas d'arbre vide. Mais les forêts (qui ne sont rien d'autre que des listes d'arbres) peuvent être vides. Par conséquent un objet de la classe ***Tree*** ne sera jamais **null**, mais un objet de la classe ***Forest*** si. Notons que les enfants d'une feuille d'un arbre sont effectivement une forêt vide.
2. Pour un arbre :
$$\mathcal{PO}(n, F) = \mathcal{PO}(F)\ n$$
Et une forêt :
$$\mathcal{PO}(\epsilon) = \epsilon \ \text{ (liste vide)}, \qquad \mathcal{PO}(T, F) = \mathcal{PO}(T)\ \mathcal{PO}(F)$$
3. En théorie, nous avons pris quelques libertés avec les listes de sommets. En pratique, il faut concaténer nos amies les listes. La possibilité de forêt vide conduit à choisir une méthode statique pour les forêts. Ce n'est pas obligatoire, mais le code est alors particulièrement compact.

```
class Tree {
  ...
  List postOrder() {
    return List.concat(Forest.postOrder(children), new List (node,null)) ;
  }
  ...
}

class Forest {
  ...
  static List postOrder(Forest f) {
    return List.concat(first.postOrder(), postOrder(next)) ;
  }
  ...
}
```

En fait, on peut se passer des concaténations en ajoutant un argument aux méthodes `postOrder`.

```
class Tree {
  ...
  List postOrder(List k) {
    return Forest.postOrder(children, new List(node,k)) ;
  }

  List postOrder() {
    return postOrder(null) ;
  }
  ...
}

class Forest {
  ...
  static List postOrder(Forest f, List k) {
    if (f == null)
      return k ;
    else
      return f.first.postOrder(postOrder(f.next,k)) ;
```

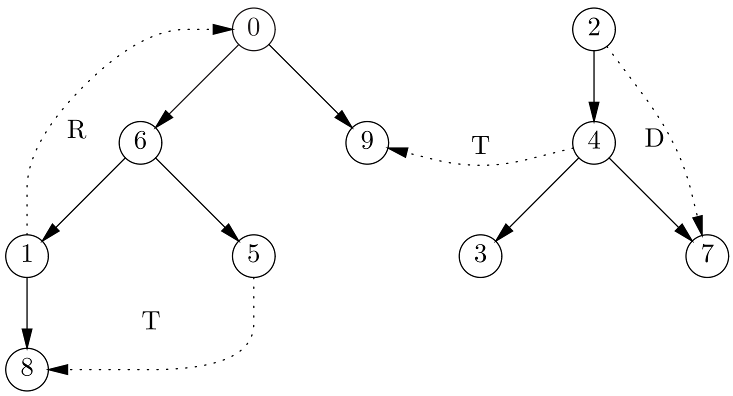
```
  }

  static List postOrder(Forest f) {
    return postOrder(f,null) ;
  }
  ...
}
```

On notera l'utilisation de la surcharge pour définir, dans la même classe deux méthodes homonymes, mais distinctes.

Exercice 4

1. Un appel à `dff` renvoie une forêt, telle que produite par un parcours en profondeur d'abord du graphe.
2. Et voilà, les arcs du graphe qui ne sont pas des arcs de l'arbre sont indiqués et identifiés.



3. Les arcs transverses etc. du graphe sont définis dans le poly, figure 1.10. On peut paraphraser cette définition en termes de forêt, il y a donc des arcs de la forêt, des arcs de descente (qui joignent un sommet à un de ses descendants dans la forêt, sans être un arc de l'arbre), des arcs de retour (qui joignent un sommet à un de ses ancêtres) et des arcs transverses (les autres !).

   Les arcs transverses vont toujours de la droite vers la gauche sur le dessin. Cette propriété peut être considérée comme « *évidente* ». Détaillons un peu. De fait, par définition de l'arc transverse, n n'est pas un ancêtre de m ni m un ancêtre de n. Il y a alors deux cas.
   - n et m n'ont pas d'ancêtre en commun, alors n et m appartiennent à deux arbres distincts de la forêt, l'arbre qui contient n étant nécessairement placé à droite de celui de m.
   - n et m ont au moins un ancêtre commun, soit a l'ancêtre commun le plus profond, alors les sous-arbres de a constituent une forêt, et le raisonnement précédent s'applique.
4. Une définition de l'ordre préfixe. Pour un arbre :

$$\mathcal{PR}(n, F) = n\ \mathcal{PR}(F)$$

   Et une forêt :

$$\mathcal{PR}(\epsilon) = \epsilon \ \ \text{(liste vide)}, \qquad \mathcal{PR}(T, F) = \mathcal{PR}(T)\ \mathcal{PR}(F)$$

5. On remarque que les arcs de l'arbre et de descente ne sont pas discriminables à l'aide des ordres et que les arcs de retour sont les seuls dont la source précède la cible selon l'ordre

postfixe.

$n \to m$	préfixe	postfixe
F	$\leq$	$\geq$
D	$\leq$	$\geq$
R	$\geq$	$\leq$
T	$\geq$	$\geq$

On note aussi que la propriété des arcs transverses est bien visible.

6. Montrons qu'il y a des arcs de retour si et seulement si le graphe contient un cycle. Le sens direct est facile, puisque par définition il existe d'une part l'arc de retour $n \to m$ et d'autre part un chemin de l'arbre de l'ancêtre m à son descendant n.

   Réciproquement, soit un cycle $n \to m \to \ldots \to n$. Montrons alors que l'un des arcs du cycle est de retour. Sans perte de généralité, n est le plus petit sommet du cycle selon l'ordre postfixe et $n \to m$ est un arc de retour.

   Notons que dans le poly, les arcs de retour sont identifiés à l'aide d'un coloriage BLANC $\to$ GRIS $\to$ NOIR des sommets. Un sommet étant gris lors de la construction de l'arbre dont il est la racine (et donc de l'exploration des ses descendants).

Exercice 5

1. Il est impossible de produire un tri topologique en cas de cycle.
2. Les arcs de retour étant les seuls dont l'origine précède la cible selon l'ordre postfixe et aucun arc $n \to m$ d'un graphe acyclique ne pouvant être classsifié comme de retour, n suit m dans l'ordre postfixe. On peut conclure que si il y a chemin de $n \to \ldots \to m$, alors n suit m dans tout ordre postfixe. Soit, un ordonancement postfixe d'un graphe acyclique est l'inverse d'un tri topologique.

   ```
   List tSort() {
     return List.reverse(bff().postOrder()) ;
   }
   ```

3. Si le graphe exprime des dépendances entre des tâches à accomplir (par exemple mettre ses chaussettes avant ses chaussures...), alors le tri topologique donne un ordonnancement des tâches compatible avec les dépendances entre tâches.

   En informatique, le cas se présente souvent lors de la compilation et recompilation d'un programme dont le source est réparti entre plusieurs fichiers. Unix fournit un utilitaire `make` qui dont le principe est de produire un tri topologique. La version web du cours vous invite à découvrir l'écriture d'un genre de `make` en Java.