INF 431, COMPOSITION D'INFORMATIQUE

Luc Maranget et Nicolas Sendrier

2 mai 2007

⋆ ⋆ ⋆

Partie I, Ensembles de chaînes

Le but de cette partie est la réalisation d'une classe **StringSet** des ensembles de chaînes qui suit (en partie) le modèle des `Set<String>` de la bibliothèque de Java.

On suppose déjà écrite une classe simple des cellules de liste de chaînes.

```
class List {
  String val ; List next ;

  List (String val, List next) {
    this.val = val ; this.next = next ;
  }
}
```

Cette classe **List** sera utilisée telle quelle dans la suite du problème, ou complétée si besoin est. Autrement dit, vous ne pouvez pas supposer que la classe **List** offre des méthodes sans les écrire vous-même.

Voici un squelette de la classe **StringSet**.

```
class StringSet implements Iterable<String> {
  // Liste cachée des éléments.
  private List p ;
  // Constructeur de l'ensemble vide.
  StringSet () { p = null ; }

  boolean isEmpty() { ... } // Vide, ou pas ?
  boolean contains(String x) { ... } // Contient x ou pas ?
  String choose() { ... } // Choisir un élément arbitraire.
  boolean add(String x) { ... } // Ajouter x.
  boolean remove(String x) { ... } // Enlever x.
  public Iterator<String> iterator()  { ... } // Renvoie un itérateur.
}
```

Chaque objet de la classe **StringSet** représente un ensemble géré sur le mode destructif. Il possède en particulier un champ privé **List** `p` qui est la liste de ses éléments, *sans doublons*. Tous les corps de méthodes notés `{...}` ci-dessus sont à écrire comme demandé par les questions qui suivent.

Question 1. Écrire les deux premières méthodes, `isEmpty` et `contains`.

Réponse :

```
  boolean isEmpty() { return p == null ; }

  boolean contains(String x) {
    for (List q = p ; q != null ; q = q.next)
      if (x.equals(q.val)) return true ;
    return false ;
  }
```

□

Question 2. Écrire la méthode `choose`. Si l'ensemble est vide, la méthode `choose` doit lancer l'exception `NoSuchElementException`. Sinon, l'appel à `choose` renvoie un élément, peu importe lequel, de l'ensemble. L'ensemble n'est pas modifié.

Réponse :

```
String choose() {
  if (p == null) throw new NoSuchElementException () ;
  return p.val ;
}
```

□

Question 3. Écrire la méthode `add`. Soient `xs`, un objet **StringSet**, et `x`, une chaîne. Si `x` n'est pas déjà présent dans `xs`, alors l'appel `xs.add(x)` ajoute l'élément `x` à l'ensemble `xs` (qui est donc modifié) et renvoie **true**. Sinon, `xs` est inchangé, et l'appel renvoie **false**.

Réponse :

```
boolean add(String x) {
  if (contains(x)) return false ;
  p = new List (x, p) ;
  return true ;
}
```

□

Question 4. Écrire la méthode `remove` qui est semblable à `add`, à ceci près que l'élément est supprimé de l'ensemble. Ici encore le booléen renvoyé traduit un changement de l'ensemble — **true** si l'élément a été effectivement supprimé, **false** sinon.

Réponse :

```
boolean remove(String x) {
  if (p == null) return false ;
  if (p.val.equals(x)) {
    p = p.next ;
    return true ;
  }
  List prev = p ;
  for (List q = p.next ; q != null ; q = q.next) {
    if (q.val.equals(x)) {
      prev.next = q.next ;
      return true ;
    }
    prev = prev.next ;
  }
  return false ;
}
```

Une solution récursive est un peu plus simple.

```
// NB: on suppose que x est dans la list p, donc par hypothèse p != null
private static List removeList(String x, List p) {
  if (x.equals(p.val)) {
    return p.next ;
  } else {
    p.next = removeList(x, p.next) ;
    return p ;
  }
}
```

```
boolean remove(x) {
  if (contains(x)) {
    p = removeList(x, p) ;
    return true ;
  } else {
    return false ;
  }
}
```

Certes il y a jusqu'à deux parcours de liste, mais la complexité est identique (linéaire en la longueur de la liste).

□

Question 5. Majorer le mieux possible l'ordre de grandeur asymptotique (notation O) des opérations élémentaires sur les ensembles, `contains`, `add` et `remove`, en fonction du cardinal de l'ensemble. Quelle autre structure de données que la liste pourriez vous utiliser pour améliorer ces coûts ?

Réponse : C'est une question de cours. Au pire les trois méthodes parcourent toute la liste `p`. Elles sont donc en $O(n)$.

Si à la place de la liste `p` on emploie un arbre binaire de recherche *équilibré* (AVL par exemple), alors les coûts se réduisent en $O(\log(n))$. Avec une table de hachage, on obtient des coûts réputés en temps constant — sous réserve de hachage uniforme et de table bien dimensionnée. □

Question 6. Écrire la méthode `iterator`, cette méthode renvoie un objet qui implémente l'interface `Iterator<String>`. L'interface générique `Iterator<E>` est fournie par la bibliothèque de Java, en voici une version simplifiée.

```
public interface Iterator<E> {
  public boolean hasNext() ;
  public E next () ;
}
```

L'objet renvoyé par `iterator()` possède donc les méthodes **public boolean** `hasNext()` et **public** `String next()`, qui servent à parcourir les éléments de l'ensemble. La première méthode teste si le parcours est terminé, tandis que la seconde renvoie un élément et avance le parcours d'un cran. La combinaison de ces deux méthodes permet de parcourir tous les éléments d'un ensemble, pour par exemple les afficher.

```
StringSet xs = ... ;
Iterator<String> it = xs.iterator() ;
while (it.hasNext()) {
   String x = it.next() ;
   System.out.println(x) ;
}
```

Réponse : On définit d'abord la classe des itérateurs sur les ensembles de chaînes. Il suffit de garder une référence (champ `current` ci-dessous) sur la liste des éléments, référence qui progresse à chaque appel à `next()`.

```
class StringSetIterator implements Iterator<String> {
  private List current ;

  StringSetIterator (List p) {
    current = p ;
  }
```

```
  public boolean hasNext() { return current != null ; }

  public String next() {
    String r = current.val ;
    current = current.next ;
    return r ;
  }
}
```

Ensuite la méthode `iterator` d'un objet **StringSet** se contente de renvoyer un nouvel objet **StringSetIterator** , initialisé au début de sa liste privée `p`.

```
class StringSet ...

  public Iterator<String> iterator() {
    return new StringSetIterator (p) ;
  }
  ...
```

□

Notre classe **StringSet** implémente l'interface `Iterable<String>`. Cela autorise une écriture simplifiée des parcours d'ensemble. Par exemple, l'affichage d'un ensemble peut aussi s'écrire ainsi :

```
StringSet xs = ... ;
for (String x : xs ) {
   System.out.println(x) ;
}
```

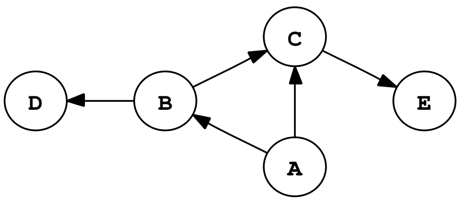
On ne se privera pas d'employer l'écriture simplifiée dans la suite du problème.

Partie II, Exécution des paquets logiciels.

Les divers logiciels que l'on peut installer sur une machine sont définis comme des paquets. Un paquet est concrètement un regroupement de programmes, de bibliothèques, etc.

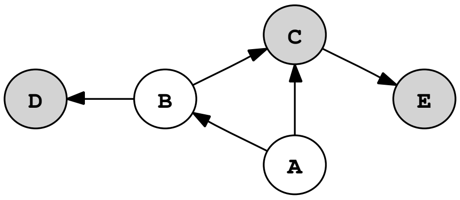
Pour fonctionner correctement les paquets ont la plupart du temps besoin d'exécuter d'autres paquets. On dit alors qu'un paquet *dépend pour son exécution* d'un autre. Par exemple, le compilateur Java (paquet `sun-java5-jdk`) est un programme Java. Son exécution entraîne directement l'exécution de la machine virtuelle Java (paquet `sun-java-bin`) et le compilateur peut directement faire appel aux méthodes de la bibliothèque Java (paquet `sun-java-jre`).

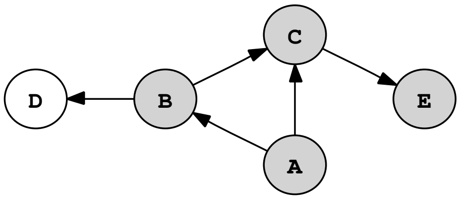
L'ensemble des dépendances entre paquets est idéalement représenté par un graphe orienté, dit graphe des dépendances. Les sommets du graphe sont les noms des paquets et les arcs traduisent les dépendances pour l'exécution. Par exemple :



Il est important de comprendre que pour exécuter le paquet `A`, on a besoin de tous les paquets de `A` à `E`. En effet, non seulement, `A` dépend de `B` et de `C`. Mais aussi, `B` et `C` dépendent de `D` et de `E` respectivement.

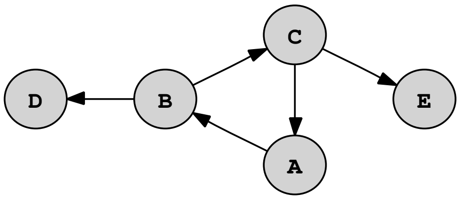
Les paquets peuvent être installés sur la machine ou pas. Dans les dessins, les paquets installés sont grisés. Voici deux états de machine.





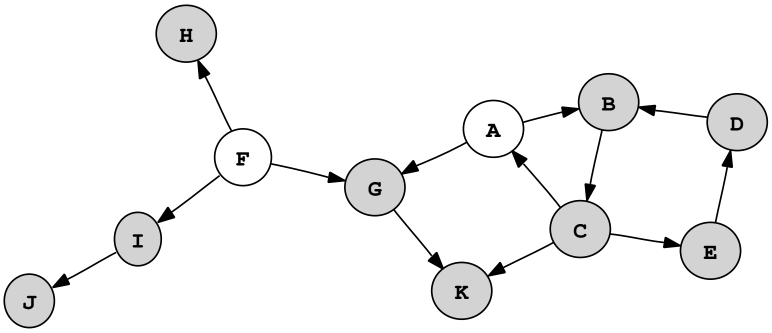
Un paquet est *exécutable*, quand il est installé et que tous les paquets dont il besoin pour s'exécuter sont installés. Un état de la machine est *cohérent* quand tous les paquets installés sont exécutables. Dans l'exemple de gauche, l'état de la machine est cohérent. En revanche, l'état de droite n'est pas cohérent, car les paquets `B` et `A` ne sont pas exécutables — le paquet `B` directement, et le paquet `A` indirectement.

Il faut noter que les dépendances circulaires ne posent pas de problème.



Ici, tous les paquets sont installés et tous sont exécutables.

Question 7. Donner tous les paquets exécutables de l'état suivant. Cet état est-il cohérent ?



Réponse : Sont exécutables, `J`, `I`, `H`, `K` et `G`. L'état n'est pas cohérent car, par exemple, `C` n'est pas exécutable, puisqu'il dépend de `A` qui n'est pas installé. □

Nous représentons en Java l'état d'une machine par la classe **Mach**.

```
class Mach {
  // Ensemble des paquets installés.
  static StringSet installed ;
  // Renvoie l'ensemble des paquets dont pack dépend.
  static StringSet dependsRun(String pack)
  ...
```

Le graphe des dépendances est représenté par la méthode `dependsRun` ci-dessus, qui prend un nom de paquet `pack` en argument et renvoie l'ensemble (**StringSet**) des paquets dont `pack` dépend pour son exécution. La méthode `dependsRun` est donnée, vous n'avez pas à l'écrire. Toutes les méthodes demandées par la suite sont des méthodes statiques de la classe **Mach**.

Question 8. Écrire une méthode **static StringSet** `neededToRun(String pack)` qui renvoie l'ensemble des paquets dont `pack` a besoin pour s'exécuter. Majorer le nombre d'opérations `add` effectuées, pour un graphe comportant n sommets et m arcs.

Réponse : Parcours de graphe écrit avec les moyens du bord, c'est-à-dire avec des ensembles et non pas des marques.

```
static StringSet neededToRun(String pack) {
  StringSet work = new StringSet () ;   // Sommet actifs
  StringSet needed = new StringSet () ; // Sommets vus

  needed.add(pack) ; work.add(pack) ;
  while (!work.isEmpty()) {
    String p = work.choose() ;
    work.remove(p) ;
    for (String d : dependsRun(p)) {
      if (needed.add(d)) { // non vu -> vu
        work.add(d) ;
      }
    }
  }
  return needed ;
}
```

L'appel `work.add` est effectué au plus n fois, car un sommet donné n'est ajouté à `work` qu'au plus une fois (grâce à l'ensemble `needed` qui fait office d'ensemble des sommets vus).

L'appel `needed.add` est effectué au plus $1 + m$ fois. On a d'une part l'appel initial, et d'autre part un appel par arc rencontré. Or les sources des arcs rencontrés sont deux à deux distinctes (cf. ci-dessus).

Ou aurait pu remplacer l'ensemble `work` par une pile, une file, etc. On serait alors un peu plus efficace, car lors de l'ajout de `d` à `work`, nous sommes sûrs que `d` n'appartient pas à `work`.

□

Question 9. Écrire deux méthodes.
a) La méthode **static StringSet** `missingPackages(String pack)` renvoie l'ensemble des paquets dont `pack` a besoin pour s'exécuter et qui ne sont pas déjà installés.

Réponse : Il faut tout simplement enlever les paquets installés des paquets dont `pack` a besoin pour s'exécuter. Compte tenu de ce que nous savons de l'implémentation de **StringSet** il est plus que raisonnable de fabriquer un nouvel ensemble, au lieu d'enlever les éléments de `needed` ci-dessous à l'aide de sa méthode `remove`.

```
static StringSet missingPackages(String pack) {
  StringSet needed = neededToRun(pack) ;
  StringSet r = new StringSet () ;
  for (String n : needed) {
    if (!installed.contains(n)) r.add(n) ;
  }
  return r ;
}
```

□

b) La méthode **static boolean** `canRun(String pack)` détermine si `pack` est exécutable ou pas.

Réponse :

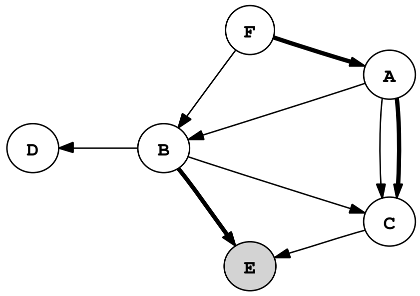
```
static boolean canRun(String pack) {
  return missingPackages(pack).isEmpty() ;
}
```

□

Partie III, Installation des paquets logiciels.

L'utilisateur de la machine peut souhaiter installer de nouveaux paquets. Il est logique de n'installer que des paquets exécutables, et le système d'exploitation s'efforce de conserver la machine dans un état cohérent. Plus précisément, quand l'utilisateur demande d'installer un paquet a, le système d'exploitation de la machine décide d'installer tous les paquets de l'ensemble `missingPackages` (a). On dit alors que le système d'exploitation installe le paquet *a complètement*.

Mais les choses ne sont pas si simples. L'installation d'un paquet a se décompose en deux étapes : d'abord obtention du paquet a (téléchargement, CD-ROM, etc.) ; puis installation proprement dite de a (copie des fichiers à leur place définitive, configuration, etc.). Il peut se faire que cette dernière procédure exécute un autre paquet b. En ce cas, nous dirons que le paquet *a dépend pour son installation* du paquet b. Si a dépend pour son installation de b et que b n'est pas exécutable, alors l'installation de *a échoue*. Nous notons $a \rightarrow b$ (flèche ordinaire) une dépendance d'exécution et $a \boldsymbol{\longrightarrow} b$ (flèche en gras) une dépendance d'installation Le graphe des dépendances comprend donc désormais deux sortes d'arcs. On aura par exemple :



Question 10. La machine est dans l'état décrit juste ci-dessus. En réponse à une demande d'installation de `F`, le système d'exploitation décide d'installer `A`, `B`, `C`, `D` et `F`, dans cet ordre. Identifier un problème et y remédier.

Réponse : L'installation complète échoue d'entrée de jeu, puisque l'installation de `A` demande d'exécuter `C` et que l'on essaie d'installer `A` avant `C`.

On peut réussir à installer complètement `F` en installant les paquets dans l'ordre, `C`, `B`, `D`, `A` et puis `F`. Plus généralement, est correct tout ordre qui installe `C` avant `A` et installe complètement `A` avant `F`. □

Question 11. Nous notons I l'ensemble des paquets installés, que nous supposons cohérent. Nous notons $a \Rightarrow b$, lorsque $a \rightarrow b$, ou $a \boldsymbol{\longrightarrow} b$, ou les deux. Nous notons $\Rightarrow^*$ la fermeture réflexive et transitive de $\Rightarrow$ — $a \Rightarrow^* b$ signifie qu'il existe un chemin (éventuellement vide) de a vers b.

Pour tout paquet a *non-installé*, nous définissons l'ensemble d'arcs

$$\mathrm{Dep}(a, I) = \{b \Rightarrow c \mid c \notin I, a \Rightarrow^* b \Rightarrow c\}.$$

L'ensemble $\mathrm{Dep}(a, I)$ est assimilable à un sous-graphe du graphe des dépendances, limité aux paquets non-encore installés et nécessaires pour installer et exécuter a. Par cohérence de I, le graphe $\mathrm{Dep}(a, I)$ est connexe. Nous définissons également $\Delta(a, I)$, l'ensemble des sommets du sous-graphe $\mathrm{Dep}(a, I)$.

$$\Delta(a, I) = \{b \mid \exists\, c \Rightarrow b \in \mathrm{Dep}(a, I)\} \cup \{\, a \,\}$$

a) On suppose que $\mathrm{Dep}(a, I)$ contient un cycle de la forme $b \boldsymbol{\longrightarrow} c \Rightarrow^* b$. Expliquer pourquoi l'installation complète de a est impossible dans ces conditions.

Réponse : Installer complètement a demande d'installer b. Or, b est impossible à installer. En effet, installer b demande d'exécuter c, et c n'est pas exécutable avant que b ne soit installé. □

b) On suppose qu'il n'existe pas de cycle $b \twoheadrightarrow c \Rightarrow^* b$ dans $\mathrm{Dep}(a, I)$. Montrer qu'il existe alors un paquet d de $\Delta(a, I)$ qui est tel que $\mathrm{Dep}(d, I)$ ne contient pas d'arc $\twoheadrightarrow$. On demande un algorithme exprimé en pseudo-code qui étant donné a calcule d.

Réponse : Voici l'algorithme :

```
choose(a,I)
  tantque ∃(c ➞ b) ∈ Dep(a,I)
    a <- b
  retourner a
```

Il est évident que l'algorithme, s'il termine, renvoie un paquet b avec $\mathrm{Dep}(b, I)$ ne contient pas d'arc $\twoheadrightarrow$. L'algorithme termine car l'ensemble $\mathrm{Dep}(a, I)$ contient strictement $\mathrm{Dep}(b, I)$, aucun arc $a' \Rightarrow a$ (*i.e.*, aboutissant en a) ne pouvant appartenir à $\mathrm{Dep}(b, I)$ — sinon, il y aurait un cycle contenant $c \twoheadrightarrow b$. $\square$

c) En déduire un algorithme qui installe a complètement et sans échec, dans le cas précité où il n'existe pas de cycle $b \twoheadrightarrow c \Rightarrow^* b$ dans $\mathrm{Dep}(a, I)$.

Réponse : Le paquet d solution de la question précédente s'installe complètement sans échec, en installant tous les paquets de $\Delta(d, I)$ dans n'importe quel ordre. Ici intervient la cohérence : si l'installation de $e \in \Delta(d, I)$ a besoin d'exécuter un paquet f (si $e \twoheadrightarrow f$), alors f est nécessairement installé et est donc exécutable. Notons finalement que l'état produit est cohérent, une fois installés tous les paquets de $\Delta(d, I)$.

On peut ensuite installer tous les paquets de $\Delta(a, I)$ ainsi.

```
install(a)
  d <- choose(a,I)
  tantque a ≠ d
    I <- Δ(d,I) ∪ I
    d <- choose(a,I)
  I <- Δ(a,I) ∪ I
```

Où les affectations de I expriment les installations effectives. Les installations effectuées ne peuvent pas échouer (cf. ci-dessus). L'algorithme termine, car $\Delta(a, I)$ décroît strictement à chaque tour de boucle (d au moins est installé). $\square$

Les dépendances pour l'installation sont représentées en machine par une nouvelle méthode statique `dependsInstall` de la classe **Mach**. L'installation effective d'un paquet est réalisée par la méthode `install`.

```java
// Renvoie l'ensemble des paquets dont pack dépend pour son installation.
static StringSet dependsInstall(String pack)
// Installer effectivement pack,
static void install(String pack) {
  for (String d : dependsInstall(pack))
    if (!canRun(d)) throw new Error ("Échec") ;
  installed.add(pack) ;
}
```

Ici encore, la méthode `dependsInstall` est donnée, vous n'avez pas à l'écrire. La méthode `install` est également donnée, son code traduit en Java ce que l'énoncé entend par l'échec de l'installation d'un paquet.

Question 12. Écrire une méthode **static void** `safeInstall(String pack)` qui installe `pack` complètement si c'est possible, ou sinon provoque une erreur. Par ailleurs, `safeInstall` doit être aussi efficace que possible, c'est-à-dire effectuer au plus un parcours du graphe des dépendances. Aucune preuve de correction n'est demandée.

Réponse : Un parcours en profondeur d'abord permet l'installation sûre.

```
static void safeInstall(String pack) {
  topo(pack, new StringSet ()) ;
}

static void topo(String p, StringSet seen) {
  if (seen.contains(p) || installed.contains(p)) return ;
  seen.add(p) ;
  for (String s : dependsInstall(p)) topo(s, seen) ;
  for (String d : dependsRun(p)) topo(d, seen) ;
  install(p) ;
}
```

Sans faire de preuve complète, le parcours en profondeur d'abord permet l'installation sans échec si possible. L'installation échoue si un `s` n'est pas exécutable lors de l'exécution de `install(p)`. En raison des propriétés du parcours en profondeur, cet évènement survient

- Si l'arc `p` $\rightarrow$ `s` est un arc de retour.
- Ou si `p` $\rightarrow$ `s` est un arc de l'arbre de recouvrement, et que $\mathrm{Dep}(\mathtt{s}, I)$ contient un arc de retour pointant sur un ancêtre de `p` au sens large.

Dans les deux cas, le test d'appartenance à `seen` conduit à revenir de `topo` sans que le paquet cible de l'arc de retour soit installé. Ce qui conduit plus tard à un erreur lors de la tenative d'installation de `p`. Ce qui n'invalide pas la correction de `safeInstall`, puisque dans les deux cas, il existe un cycle qui interdit l'installation de `p`.

□