UNIVERSITÉ PARIS SUD

Projet de Combinatoire

Master d'informatique

—Génération Combinatoire—

Génération d'objets combinatoires décrit par une grammaire

Le but de ce projet est de compter et d'engendrer l'ensemble des objets combinatoires étiqueté décrits par une grammaire. Il est ainsi possible d'engendrer une grande variété d'objets comme des ensembles, des arbres ou des mots.

Le projet sera implanté en `Python`. On pourra travailler seul ou en binôme. La date de remise sera précisée ultérieurement. Toutes les fonctions de ce projet devront être commentées et testées.
On rédigera également un **rapport** présentant les fonctionnalités et répondant aux questions théoriques du sujet. Les algorithmes et choix d'implantations devront être expliqués.

---

1 Introduction : quelques exemples

1.1 Sequences simples, triées, cycliques et zig-zag

Dans ce projet, nous allons construire et implémenter des grammaires pour décrire des objets **étiquetés**. Par rapport à ce que l'on a vu en cours, on ne parlera pas de l'ensemble des objets de taille n, mais de **l'ensemble des objets étiquetés par les éléments de** S. L'ensemble des étiquettes S sera décrit par une **liste supposée trié et sans doublons**. Un objet décrit par la grammaire devra contenir **toutes les étiquettes une fois et une seule**. Avant de décrire les grammaires nous commençons par quelques exemples :

1. Séquences simples : les objets de ce type sont les listes contenants toutes les étiquettes. Ce sont donc les permutations de l'entrée S. Ainsi

   ```
   Seq.list([1,2,3]) = [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]
   ```

2. Séquence ordonnées : les objets de ce type sont les listes contenants toutes les étiquettes dans l'ordre. Il n'y a donc qu'un seul objet : la liste donnée en entrée :

   ```
   SortedSeq.list([1,2,3]) = [[1,2,3]]
   ```

3. Séquence cyclique : les objets de ce type sont les listes à permutation cyclique prêt. On peut donc supposer que le plus petit élément est au début :

   ```
   Cycle.list([1,2,3]) = [[1,2,3],[1,3,2]]
   Cycle.list([0,1,2,3]) =
           [[0,1,2,3],[0,1,3,2],[0,2,1,3],[0,2,3,1],[0,3,1,2],[0,3,2,1]
   ```

4. Séquence zig-zag : les objets de ce type sont les listes $(a_0 < a_1 > a_2 < a_3 > a_4 < \dots)$:

   ```
   ZigZag.list([1,2,3]) = [[1,3,2],[2,3,1]]
   ZigZag.list([1,2,3,4]) =
           [[1,3,2,4],[1,4,2,3],[2,3,1,4],[2,4,1,3],[3,4,1,2]]
   ```

1.2 Les arbres binaires

Un **arbre binaire étiqueté** est soit une feuille, soit un noeud avec une étiquette sur lequel on a greffé deux arbres binaires. Dans la suite de ce sujet, on suppose les arbres déjà implanté avec un constructeur `Node(e, l, r)` pour les noeuds et une constante `Leaf` pour les feuilles.

```
>>> tr = Node(1, Leaf, Node(2, Leaf, Leaf))
```

On peut ainsi décrire l'ensemble des arbres binaires par les définitions récursives suivantes :

- l'ensemble `"Trees"` des arbres est la réunion disjointe (que l'on notera en `Python` par le constructeur `UnionRule`) de deux ensembles : l'ensemble `"Nodes"` des noeuds et l'ensemble `"Leaf"` des feuilles ;
- l'ensemble `"Nodes"` des noeuds est obtenu en ajoutant une étiquette à une paire d'arbres (c'est-à-dire un élément du produit cartésien de l'ensemble des arbres avec lui même). On note `ProductRule` le constructeur du produite cartésien. Le constructeur de l'ensemble singleton formé d'une étiquette est noté `SingletonRule`
- il n'y a qu'un seul arbre possible constitué d'une feuille. C'est le un objet vide (constructeur `EpsilonRule`) `"Leaf"`.

Une telle définition est appelée **grammaire**. On écrira en `Python` la grammaire des arbres de la manière suivante :

```
ifdun = lambda x : x
treeGram = {"Tree" : UnionRule("Node", "Leaf"),
            "Node" : ProductRule("Label", "Pair",
                                 lambda (l, p) : Node(l, p[0], p[1])),
            "Pair" : ProductRule("Tree", "Tree", idfun),
            "Label" : SingletonRule(idfun),
            "Leaf" : EpsilonRule(Leaf)}
```

Le but de ce projet est d'implanter un algorithme permettant de compter, d'engendrer automatiquement la liste ainsi que de tirer au hasard un des objets décrit par une grammaire de ce type. Il y a 30 arbres possibles avec les étiqettes `'a'`, `'b'`, et `'c'` :

```
>>> treeGram["Tree"].list(range(3))
30
```

En voici quelques un :

```
>>> treeGram["Tree"].list(range(3))
[Node(0, Leaf, Node(1, Leaf, Node(2, Leaf, Leaf))),
 Node(0, Leaf, Node(1, Node(2, Leaf, Leaf), Leaf)),
 Node(0, Leaf, Node(2, Leaf, Node(1, Leaf, Leaf))),
 [...]
 Node(2, Leaf, Node(0, Leaf, Node(1, Leaf, Leaf))),
 Node(2, Leaf, Node(0, Node(1, Leaf, Leaf), Leaf)),
```

```
[...]
Node(2, Node(0, Leaf, Node(1, Leaf, Leaf)), Leaf),
Node(2, Node(0, Node(1, Leaf, Leaf), Leaf), Leaf),
Node(2, Node(1, Leaf, Node(0, Leaf, Leaf)), Leaf),
Node(2, Node(1, Node(0, Leaf, Leaf), Leaf), Leaf)]
```

1.3 Listes simples et triées

Pour que l'on obtienne un comportement intéressant avec les étiquettes, il faut pouvoir contrôler la leurs provenance. On va donc distinguer trois produits. Dans les exemples suivant, on suppose que `A` et `B` sont des séquences ordonnées `SortedSeq`.

1. le **produit cartésien ordinaire** noté `Prod`. Les étiquettes se répartissent alors de toutes les manières possibles. Ainsi, pour la règle `C = Prod('A', 'B', f)`, On a `C.list([1, 2, 3)` sont

   ```
   C.list([1, 2, 3]) =
           f([], [1,2,3]), f([1], [2,3]), f([2], [1,3]), f([3], [1,2]),
           f([1,2], [3]),  f([1,3], [2]), f([2,3], [1]), f([1,2,3], [])
   ```

2. le **produit cartésien boxed** noté `BoxProd`. La plus petite étiquette est forcément à gauche. Ainsi,

   ```
   C.list([1, 2, 3]) =
           f([], [1,2,3]), f([1], [2,3]), f([1,2], [3]), f([1,2,3], [])
   ```

3. le **produit cartésien ordonné** noté `OrdProd`. Les étiquettes se répartissent en mettant les plus petites à gauches et les plus grandes à droite. Ainsi,

   ```
   C.list([1, 2, 3]) =
           f([1], [2,3]), f([1,2], [3]), f([1,3], [2]), f([1,2,3], [])
   ```

Avec ces définition, on peut donc définir les arbres binaires croissants par la grammaire :

```
treeGram = {"Tree" : UnionRule("Node", "Leaf"),
            "Node" : BoxProdRule("Label", "Pair",
                                 lambda (l, p) : Node(l, p[0], p[1])),
            "Label" : SingletonRule(idfun),
            "Pair" : ProductRule("Tree", "Tree", idfun),
            "Leaf" : EpsilonRule(Leaf)}
```

Les arbres binaires de recherche sont à leur tour défini par la grammaire :

```
treeGram = {"Tree" : UnionRule("Node", "Leaf"),
            "Node" : OrdProdRule("Tree", "PairLSAD",
                                 lambda (l, p) : Node(p[0], l, p[1])),
            "Label" : SingletonRule(idfun),
            "PairLSAD" : OrdProdRule("Label", "Tree", idfun),
            "Leaf" : EpsilonRule(Leaf)}
```

2 Définitions formelles

Une grammaire décrit récursivement un ensemble d'objets. Elle est constituée d'un ensemble de règles ayant chacune un nom (chaîne de caractères). Le nom d'une règle est appelé **symbole non-terminal** ou plus simplement non-terminal de la grammaire.

Une **règle de grammaire** R décrit un ensemble qui est

- soit un singleton dont le seul élément est un objet **atomique** construit en utilisant une seule étiquette.
- soit un ensemble dont le seul élément est un objet **vide** sans étiquette (par exemple la chaîne vide).
- soit l'**union de deux ensembles** décrit par deux non-terminaux N_1 et N_2 ;
- soit en bijection avec l'une des variantes du **produit cartésien de deux ensembles** décrit par deux non-terminaux N_1 et N_2 ; L'ensemble est alors construit à partir des paires d'éléments $(e_1, e_2) \in N_1 \times N_2$. Dans ce cas, il faut de plus donner à `Python` la bijection qui construit l'objet correspondant à la paire (e_1, e_2) (concaténation pour les chaînes de caractères où constructeur `Node` pour les arbres).

La **taille** ou **poids** d'un objet est le nombre d'étiquette qu'il contient. Le poids d'un élément correspondant à une paire (e_1, e_2) est donc la somme des poids de e_1 et de e_2.

À chaque non-terminal on associe la taille du plus petit objet qui en dérive. Cette taille est appelé **valuation** du non-terminal.

2.1 Structures de données

On modélise chacune de ces classes combinatoires décrites récursivement par un objet (au sens de la programmation orientées objets) `Python`. Dans l'exemple des arbres binaires, l'objet `treeGram["Tree"]` modélise l'ensemble de tous les arbres binaires. Cet ensemble est construit comme l'union de deux sous ensembles modélisés par les objets `treeGram["Node"]` et `treeGram["Leaf"]`. La classe de l'objet `treeGram["Tree"]` est ainsi `UnionRule`, il est construit grâce à un appel au constructeur par `UnionRule("Node", "Leaf")`.

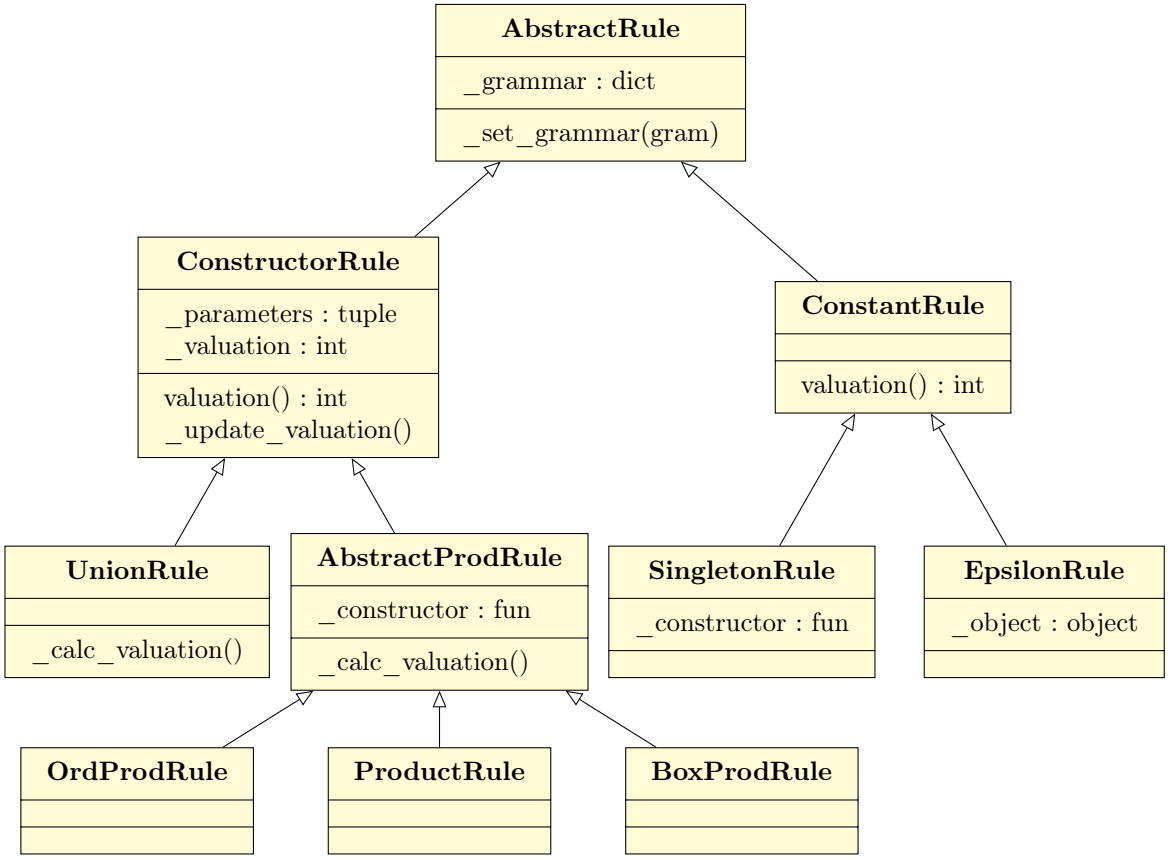
Une grammaire sera stockée sous la forme d'un dictionnaire qui associe un objet à une chaîne de caractère. Dans le but de ne pas recopier plusieurs fois du code, on utilisera avantageusement la programmation objet et l'héritage. Ainsi chaque règle de grammaire sera un objet de la classe abstraite `AbstractRule`.

— les règles de constructions qui sont les objets des classes `UnionRule` et `ProductRule` qui dérivent de la classe abstraite `ConstructorRule` ;

— les règles de constantes qui sont les objets des classes `SingletonRule` et `EpsilonRule` ;

On aura aussi avantage à faire une classe `AbstractProductRule` pour les produits abstraits dont dériverons les trois classes de produits concrets.

Voici un schéma de la hiérarchie de classe :



Voici la liste des constructeurs ou méthode d'initialisation (méthodes `__init__` en Python) des différentes classes avec les paramètres et leurs types :

- `SingletonRule.__init__(self, cons)` où `cons` est une fonction qui construit un `object` python à partir d'une étiquette ;
- `EpsilonRule.__init__(self, obj)` où `obj` est un `object` python quelconque.
- `UnionRule.__init__(self, fst, snd)` où `fst` et `snd` sont deux non terminaux (de type `Python string`) ;
- `AbstractProductRule.__init__(self, fst, snd, cons)` où `fst` et `snd` sont deux non terminaux et `cons` est une fonction qui prend un couple d'`object` et qui retourne un `object`. L'initialisation des trois produits `OrdPRule`, `BoxProdRule`, `OrdProdRule` étant identiques, on poura implémenter la méthode une seule fois dans la classe abstraite `AbstractProductRule`.

En plus des méthodes listées dans ce diagramme, chaque classe devra implanter (ou bien hériter) les méthodes d'énumération suivantes : `count, list, unrank, random`,...

Le principal problème d'implantation provient du fait qu'une grammaire est un **ensemble de définitions mutuellement récursives**. Il y a donc un travail à faire pour « casser des boucles infinies » et s'assurer que la récursion est bien fondée. Voici quelques éléments permettant de résoudre ce problème :

- Pour les règles constructeur par exemple `UnionRule("Node", "Leaf")` les sous règles (ici : `"Node"` et `"Leaf"`) ne sont connues que par les chaînes de caractères qui représentent les symboles non terminaux. Pour pouvoir associer ces chaînes aux objets associés, il faut que l'objet `UnionRule("Node", "Leaf")` ait accès à la grammaire.
- Au moment de la construction d'une règle, la grammaire qui va contenir la règle n'existe pas encore ; il faut donc attendre que la grammaire soit complètement créée pour appeler la méthode `_set_grammar` sur chaque règle. C'est le rôle de la fonction `init_grammar`.
- La fonction `init_grammar` se charge également de calculer les valuations selon l'algorithme décrit plus bas.

2.2 À rendre sur papier

ToDo: Update

1. Pour les grammaires des arbres et des mots de Fibonnacci, donner dans un tableau pour $n = 0, \dots, 10$ les réponses attendue pour la méthode `count` pour les 8 non terminaux,
2. Donner la grammaire de tous les mots sur l'alphabet `A,B`.
3. Donner la grammaire des mots de Dyck, c'est-à-dire les mots sur l'alphabet `{(,)}` et qui sont correctement parenthésés.
4. Donner la grammaire de mots sur l'alphabet `A,B` qui n'ont pas trois lettres consécutivement égales.
5. Donner la grammaire des palindromes sur l'alphabet `A`, `B`, même question sur l'alphabet `A,B,C`.
6. Donner la grammaire des mots sur l'alphabet `A,B` qui contiennent autant de lettres `A` que de lettres `B`.
7. Écrire une fonction qui vérifie qu'une grammaire est correcte, c'est-à-dire que chaque non-terminal apparaissant dans une règle est bien défini par la grammaire.

3 Algorithmes

On demande d'implanter les algorithmes de calculs suivants (dans ce qui suit `rule` est une règle quelconque) :

- Calcul de la valuation d'une grammaire (ce qui permet de vérifier la grammaire) ;
- méthode `rule.count(self, n)` qui calcule le nombre d'objets d'un poids donné n par la méthode naïve (le nombre d'objet ne dépend pas le la liste des étiquettes, il suffit donc de passer la taille de la liste en paramêtre) ;
- méthode `rule.list(self, S)` qui calcule la liste des objets étiqueté par `S` ;

— méthode `rule.unrank(self, S, i)` qui calcule le i-ème élément de la liste des objets étiqueté par `S`, sans calculer la liste ; Note importante : en `Python` la position dans une liste commence à 0 ;
— méthode `rule.random(self, S)` qui choisit équitablement au hasard un objet étiqueté par `S` (on utilisera `rule.unrank(self, S, i)` où i sera choisi aléatoirement).

Le calcul de la valuation est nécessaire aux étapes suivantes qui sont indépendantes. Je les ai néanmoins classé par ordre croissant de difficulté.

3.1 Calcul de la valuation

La **valuation** du non-terminal `nt` est la taille du plus petit objet qui en dérive. La valuation d'une grammaire est l'ensemble des valuations des terminaux. Elle vérifie les quatres règles suivantes :

- la valuation d'un `Singleton` est 1 ;
- la valuation d'un `Epsilon` est 0 ;
- la valuation de l'union `Union` des non-terminaux N_1 et N_2 est le minimum des valuations de N_1 et de N_2 ;
- la valuation d'un produit `Prod` des non-terminaux N_1 et N_2 est la somme des valuations de N_1 et de N_2 ;

Pour la calculer, on utilise l'algorithme de point fixe suivant. On part de la valuation V_0 (incorrecte) qui associe à chaque non-terminal la valeur ∞. En appliquant une fois non récursivement (pour éviter les boucles infinies) les règles précédentes à partir de V_0, on calcule une nouvelle valuation V_1. On calcule ensuite de même une valuation V_2 a partir de V_1. On recommence tant que la valuation V_n est différente de V_{n-1}. La valuation cherchée $V := V_n$ est obtenue quand $V_n = V_{n-1}$.

Note : Si $V(N) := V_n(N) = \infty$ pour un certain non terminal N, alors aucun objet ne dérive de ce non-terminal. On considère alors que la grammaire est incorrecte.

Par exemple, sur les arbres, le calcul se fait en 4 étapes :

n	`Tree`	`Leaf`	`Node`
règle :	$\min(V_{n-1}(\texttt{Leaf}), V_{n-1}(\texttt{Node}))$	1	$V_{n-1}(\texttt{Tree}) + V_{n-1}(\texttt{Tree})$
0	∞	∞	∞
1	∞	1	∞
2	1	1	∞
3	1	1	2
4	1	1	2
Final	1	1	2

3.1.1 À faire :

8. À rendre sur papier : Pour les différentes grammaire considérée, appliquer l'algorithme à la main et donner les valuations des différents non terminaux.

9. Écrire un fonction `init_grammar` qui prend en paramètre une grammaire, qui appelle sur chaque règle de la grammaire la méthode `set_grammar` et qui implante l'algorithme de calcul de la valuation.

3.2 Comptage naïf du nombre d'objets

Le comptage du nombre d'objets de poids i se fait en appliquant récursivement les règles suivantes : Soit N un non-terminal. On note $C_N(i)$ le nombre d'objet de poids i. On rappelle que le nombre de manière de répartir n étiquette en deux sous ensemble de taille k et $l = n-k$ est $\binom{n}{k} = \frac{n!}{k!\,l!}$.

- si N est un `Singleton` alors $C_N(1) = 1$ et $C_N(i) = 0$ si i est différent de 1 ;
- si N est un `Epsilon` alors alors $C_N(0) = 1$ et $C_N(i) = 0$ si i est différent de 0 ;
- si N est l'union `Union` des non-terminaux N_1 et N_2 alors

$$C_N(i) = C_{N_1}(i) + C_{N_2}(i)\,;$$

- si N est le produit `Prod` des non-terminaux N_1 et N_2

$$C_N(i) = \sum_{k+l=i} \binom{n}{k} C_{N_1}(k) \cdot C_{N_2}(l)\,;$$

- si N est le produit `OrdP` des non-terminaux N_1 et N_2

$$C_N(i) = \sum_{k+l=i} C_{N_1}(k) \cdot C_{N_2}(l)\,;$$

- si N est le produit `BoxP` des non-terminaux N_1 et N_2

$$C_N(i) = \sum_{k+l=i \text{ and } k\geq 1} \binom{n-1}{k-1} C_{N_1}(k) \cdot C_{N_2}(l)\,;$$

Pour aller plus vite et surtout **éviter des boucles infinies**, dans les produits, on ne considérera dans les sommes précédentes que les cas où $k \geq V(N_1)$ et $l \geq V(N_2)$, où $V(N)$ désigne la valuation du non-terminal N. En effet, par définition $C_N(i) = 0$ si $V(N) > i$.

3.2.1 À faire :

10. Implanter pour chaque règle de grammaire une méthode `count` qui compte le nombre d'objets d'une grammaire dérivant d'un non-terminal et d'un poids donné.

3.3 Calcul de la liste des objets

ToDo: update On applique récursivement les définitions des constructeurs `Singleton`, `Epsilon`, `Union` et `Prod` pour construire la liste des objets de taille i. En particulier, si N est le produit `Prod` des non-terminaux N_1 et N_2, la liste des objets dérivant de N et de poids i

est la concaténation des listes de tous les produits cartésiens d'éléments dérivant de N_1 et de taille k et d'éléments dérivant de N_2 et de taille l, pour tous les couples k, l tels que $k+l=i$, $k \geq V(N_1)$ et $l \geq V(N_2)$ (comme précédemment $V(M)$ désigne la valuation du non-terminal M).

Par exemple, pour obtenir les arbres de taille 3, on procède de la manière suivante

Calcul de `Tree = Union (Leaf, Node)` avec $i = 3$.

- Application de `Leaf = Singleton Leaf` avec $i = 3$, on retourne la liste vide `[]` ;
- Application de `Node = Prod(Tree, Tree)` avec $i = 3$. La valuation de `Tree` est 1. Il y a donc deux possibilités $3 = 1 + 2$ ou $3 = 2 + 1$.

  1. — Application de `Tree = Union (Leaf, Node)` avec $i = 2$.
     - `Leaf` est vide avec $i = 2$ on retourne la liste vide.
     - Application de `Node = Prod(Tree, Tree)` avec $i = 2$. La valuation de `Tree` est 1. Une seule décomposition est possible $1 + 1 \ldots$ On appelle donc deux fois `Tree` avec $i = 1 \ldots$ (Je n'écrit pas les appels récursifs)... qui retourne la liste `[Leaf]`. On retourne donc la liste

       `[Node(Leaf, Leaf)]`

     — Application de `Tree = Union (Leaf, Node)` avec $i = 1$. On retourne la liste

     `[Leaf]`

     Le produit cartésien des deux listes précédentes est la liste formée du seul élément

     `[Node(Node(Leaf, Leaf), Leaf)]`

  2. — Application de `Tree = Union (Leaf, Node)` avec $i = 1$. On retourne donc la liste

     `[Leaf]`

     — Application de `Tree = Union (Leaf, Node)` avec $i = 2$. On retourne donc la liste

     `[Node(Leaf, Leaf)]`

     Le produit cartésien des deux listes précédentes est la liste formée du seul élément

     `[Node(Leaf, Node(Leaf, Leaf))]`

  On retourne donc la liste des deux arbres :

  `[Node(Leaf, Node(Leaf, Leaf)), Node(Node(Leaf, Leaf), Leaf)]`

Pour $i = 6$, il faut essayer les décompositions $6 = 5 + 1$, $6 = 4 + 2$, $6 = 3 + 3$, $6 = 2 + 4$ et $6 = 1 + 5$. Étudions le cas $3 + 3$. Par appel récursif on trouve deux arbres de poids 3 :

`[Node(Node(Leaf, Leaf), Leaf); Node(Leaf, Node(Leaf, Leaf))]`

Le produit cartésien est donc formé de 4 éléments qui correspondent aux 4 arbres suivants :

```
Node(Node(Leaf, Node(Leaf, Leaf)), Node(Node(Leaf, Leaf), Leaf));
Node(Node(Node(Leaf, Leaf), Leaf), Node(Node(Leaf, Leaf), Leaf));
Node(Node(Leaf, Node(Leaf, Leaf)), Node(Leaf, Node(Leaf, Leaf)));
Node(Node(Node(Leaf, Leaf), Leaf), Node(Leaf, Node(Leaf, Leaf)));
```

3.4 Calcul du i-ème élément : la méthode `unrank`

Pour calculer l'élément de taille n et de rang i, on fait appel à la méthode `unrank`. Voici par exemple l'arbre de taille 6 et de rang 12.

```
>>> treeGram['Tree'].unrank(6, 12)
Node(Leaf, Node(Node(Node(Leaf, Node(Leaf, Leaf)), Leaf), Leaf))
```

Attention ! La numérotation commence à zéro.

On procédera récursivement comme suit :

— si l'on demande un objet dont le rang est supérieur au nombre d'objet on lève une exception `ValueError`.

— dans le cas `EpsilonRule` ou `SingletonRule`, on retourne l'objet.

— dans le cas d'une union : `"U" : UnionRule("A", "B")`, on suppose connu les nombres d'objets : $C_U(n) = C_A(n) + C_B(n)$. Alors l'objet de U de rang i est l'objet de A de rang i si $i < C_A(n)$ et l'objet de B de rang $i - C_A(n)$ sinon.

— dans le cas d'un produit : `"U" : ProductRule("A", "B")`, on suppose connu les nombres d'objets :

$$C_U(n) = \sum_{i=0}^{n} C_A(i)\, C_B(n-i)\,.$$

En s'inspirant de l'union, on calcule la valeur de i :

$$U(n) = \bigsqcup_{i=0}^{n} A(i) \times B(n-i)\,.$$

Il reste finalement à trouver l'élément de rang j d'un produit cartésien d'ensemble $A \times B$ où $A = \{a_1, \ldots, a_k\}$ est de cardinalité k et $B = \{b_1, \ldots, b_l\}$ est de cardinalité l. Si l'on choisi comme ordre d'énumération

$$A \times B = \{(a_1, b_1), (a_1, b_2), \ldots, (a_1, b_l), (a_2, b_1), \ldots, (a_2, b_l),\ \ldots\ , (a_k, b_1), \ldots, (a_k, b_l)\}$$

alors l'élément j est (a_r, b_q) où q et r sont respectivement le quotient et le reste de la division euclidienne de j par k.
Par exemple, pour les arbre de taille 7 :

i	0	1	2	3	4	5	6	7
$C_A(i)\ C_B(n-i)$	0	42	14	10	10	14	42	0

Ainsi, si l'on veut l'arbre de rang 73, comme $73 = 0+42+14+10+7$. On prendra $i = 4$ et l'on retournera l'arbre de rang $j = 7$ du produit cartésien $\texttt{Tree}(4) \times \texttt{Tree}(7-4)$... On a maintenant $j = 7$, $k = 5$, $l = 2$. On retournera donc l'arbre `Node(u,v)` où u est l'arbre de taille 4 et de rang 2 et v l'arbre de taille 3 de rang 1.

4 Pour rendre le programme plus sûr, efficace utilisable

4.1 Tests de cohérence génériques

Si le programme est correct, un certain nombre de propriétés doivent être vérifiées. Par exemple, la longueur du résultat de la méthode `list` doit être égale au résultat de la méthode `count`.

11. Écrire, dans le rapport, la spécification la plus précise possible que doivent vérifier les différentes méthode d'une classe combinatoire.
12. Implanter des tests génériques qui vérifie cette spécification pour toute les tailles dont la méthode `count` renvoie un résultat pas trop grand de manière à ce que les tests ne durent pas plus de quelques seconde pour une grammaire donnée.
13. Évidement, lancer les tests sur toutes les grammaires que vous aurez écrites.

4.2 Rank

Pour implanter la méthode `rank(self, obj)`, il faut que les différents non terminaux sachent analyser les objets. Il faut donc fournir au constructeur des classes les informations nécéssaires à l'analyse des objets. En particulier :

- Pour `UnionRule(A, B)`, il faut en plus que la classe ait un moyen de savoir si l'objet `obj` dérive de `A` ou de `B`.
- De même pour le produit `ProductRule(A, B)`, il faut une fonction permettant de retrouver le couple `(a, b)` à partir de `obj`.

14. Ajouter les paramètres du constructeur, attributs et méthode nécéssaire aux différentes classes pour implanter la méthode `rank(self, obj)`

4.3 Caching

15. Lors des appels récursifs, on calcule plusieurs fois la même chose. Une amélioration consiste à stocker les résultats des différents appels récursifs dans un tableau pour ne pas refaire le calcul.

4.4 Des Grammaires plus expressives

16. Pour avoir un programme plus facile à utiliser, on aimerait bien pouvoir donner des grammaires sous le format

```
{"Tree" :  Union (Singleton Leaf,
                  Prod(NonTerm "Tree", NonTerm "Tree", "".join)}
```

    Une telle grammaire est dite condensée. Une amélioration consiste à écrire une fonction qui développe automatiquement une grammaire condensée en une grammaire simple.
17. Une autre amélioration consiste à ajouté un constructeur `Bound(C, min, max)`, qui modélise l'ensemble des objets de la classe `C` dont la taille est entre `min` et `max`.

18. Ajouter le constructeur `Sequence("NonTerm", casvide, cons)` aux constructeurs autorisés. Ceci peut se faire soit dans les grammaires simples, soit dans les grammaires condensées. En effet, le constructeur `Sequence` peut s'écrire avec `Epsilon` et `Prod` :

```
"Sequence" = Union(Epsilon casvide, Prod("Sequence", "NonTerm", cons))
```

5 Pour aller plus loin

19. Écrire des grammaires pour engendrer des documents `XML` ou `HTML` complexe.

Bon Travail !