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 Grammaires étiquetées : Introduction et quelques exemples

1.1 Objets étiquetés

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.

1.1.1 Séquences

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]]
```

1.1.2 Partitions d'ensemble

Pour un ensemble S donné, les **partitions d'ensemble** sont toutes les façons de découper S en sous ensemble.

Par exemple, il y a 5 façons de partitionner l'ensemble $\{1, 2, 3\}$:

— $\{1\}, \{2\}, \{3\}$
— $\{1, 2\}, \{3\}$,
— $\{1, 3\}, \{2\}$,
— $\{1\}, \{2, 3\}$,
— $\{1, 2, 3\}$

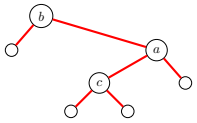
1.1.3 Arbres binaires

Les grammaires étiquetées permettent non seulement d'obtenir des séquences de nombre mais aussi des objets plus complexes tels que les arbres binaires.

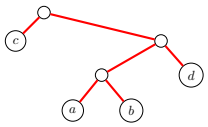
Il y a deux façons classiques d'étiqueter les arbres binaires :
— étiquetage des feuilles ;
— étiquetage des noeuds internes.

Dans le fichier `LabelledBinaryTree.py` vous trouverez une classe python très proche de celle vue en TP qui permet de représenter des arbres binaires étiquetés (sur les noeuds ou les feuilles).

Par exemple, on peut créer les arbres suivants :



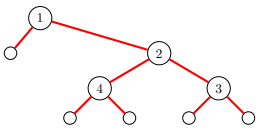

```
>>> t = Node(Leaf(), Node(Node(Leaf(),Leaf(),"c"), Leaf(), "a"), "b")
>>> t
b(leaf, a(c(leaf, leaf), leaf))
```



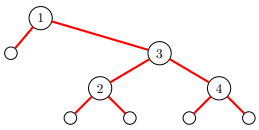

```
>>> t = Node(Leaf("c"), Node(Node(Leaf("a"),Leaf("b")), Leaf("d")))
>>> t
>>> (c, ((a, b), d))
```

Les grammaires étiquetées nous permettront par exemple d'engendrer :

— L'ensemble des arbres dont les noeuds sont étiquetés par un ensemble donné.
— L'ensemble des arbres dont les feuilles sont étiquetées par un ensemble donné.
— Les arbres croissants : l'ensemble des arbres dont les noeuds sont étiquetés par un ensemble ordonné donné tels que l'étiquette d'un noeud soit toujours inférieure aux étiquettes de ses fils.
— Les arbres binaires de recherche : l'ensemble des arbres dont les noeuds sont étiquetés par un ensemble ordonné donné tels que l'étiquette d'un noeud soit toujours supérieure à toutes les étiquettes de son fils gauche et inférieure à toutes les étiquettes de son fils droit.



Exemple arbre croissant



Exemple ABR

1.2 Définitions formelles des grammaires étiquetées

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 ou l'arbre vide).
— soit l'**union de deux ensembles** décrit par deux non-terminaux N_1 et N_2 ;
— soit une variante du **produit cartésien de deux ensembles** [(1)] décrit par deux non-terminaux N_1 et N_2 ; L'ensemble est alors construit à partir des paires d'éléments $(e_1, e_2) \in N_1 \times N_2$.

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.

Le but du projet est de permettre à un utilisateur de décrire une grammaire en python sous la forme d'un dictionnaire associant une règle à chaque non-terminal et calculant :
— Le nombre d'objet d'une certaine taille décrits par la grammaire (méthode `count`)
— La liste des objets étiquetés par une liste d'étiquettes donnée (méthode `list`)
— Un objet identifié par son rang dans la liste (méthode `unrank`)
— Le rang d'un objet donné (méthode `rank`)

(1). les variantes possibles sont décrites un peu plus tard

1.3 Quelques exemples de base (mais indispensables !)

Commençons par des exemples triviaux de grammaires. Nous vous présentons à chaque fois la description de la grammaire et son implantation python telle qu'elle devra fonctionner lorsque vous aurez réalisé le projet.

Tout le code python de déclaration des grammaires avec les tests associés se trouve dans le fichier `tests.py`.

1.3.1 La Grammaire singleton

Voici une grammaire contenant une unique règle de type `Singleton`. Cette grammaire ne peut engendrer que des objets de taille 1.

Singl $\rightarrow$ Singleton

En python, on l'implantera de cette façon :

```
GSingl = {
    "Singl": SingletonRule(lambda x:x)
}
```

Ici, l'argument `lambda x:x` est donné pour initialiser l'objet `SingletonRule`. Cela correspond à la fonction qui sera utilisée pour concrètement fabriquer l'objet combinatoire associé à une étiquette. Dans le cas de cette grammaire, c'est une fonction triviale qui prend l'étiquette (typiquement une chaîne de caractère ou un entier) en argument et la renvoie directement. On verra dans certains cas que cette fonction peut être plus complexe, par exemple dans le cas des arbres binaires.

Voilà quelques exemples d'utilisation de cette grammaire et des résultats attendus pour les méthodes `count` et `list` que vous devrez implanter. La description précise de ces méthodes et des algorithmes associés est donnée en Section 3.

```
init_grammar(GSingl)
assert GSingl["Singl"].count(0) == 0 # nb d'objets de taille 0
assert GSingl["Singl"].count(1) == 1 # nb d'objets de taille 1
assert GSingl["Singl"].count(2) == 0 # nb d'objets de taille 2

# liste des objets de taille 0
assert GSingl["Singl"].list([]) == []
# liste des objets étiquetés par un ensemble donné
assert GSingl["Singl"].list(["a"]) == ["a"]
assert GSingl["Singl"].list(["b"]) == ["b"]
assert GSingl["Singl"].list(["a","b"]) == []
```

1.3.2 La Grammaire Epsilon

Voici une grammaire contenant une unique règle de type `Epsilon`. Cette grammaire n'engendre qu'un objet unique de taille 0. Dans l'exemple suivant c'est la chaîne vide.

Empty → Epsilon(mot vide)

En python, on l'implantera de cette façon :

```
GMotVide = {
    "Empty": EpsilonRule("")
}
```

L'objet `Epsilon` prend en paramètre l'objet de taille 0 qu'il crée. Voilà quelques exemples d'utilisation de cette grammaire et des résultats attendus pour les méthodes `count` et `list` que vous devrez implanter.

```
init_grammar(GMotVide)
assert GMotVide["Empty"].count(0) == 1 # nb d'objets de taille 0
assert GMotVide["Empty"].count(1) == 0 # nb d'objets de taille 1
assert GMotVide["Empty"].count(2) == 0 # nb d'objets de taille 2

# liste des objets de taille 0
assert GMotVide["Empty"].list([]) == [""]
# liste des objets étiquetés par un ensemble donné
assert GMotVide["Empty"].list(["a"]) == []
assert GMotVide["Empty"].list(["b"]) == []
assert GMotVide["Empty"].list(["a","b"]) == []
```

1.3.3 Les permutations de taille 2

Voilà un premier exemple de grammaire à 2 règles qui engendre les permutations de taille 2.

Perms → Produit(**Letter**, **Letter**)
Letter → Singleton

En python, on l'implantera de cette façon :

```
Perm2 = {
    "Perms": ProductRule("Letter","Letter", lambda a,b: [a,b]),
    "Letter": SingletonRule(lambda x:x)
}
```

Remarquez que l'objet `ProductRule` prend en paramètre une fonction qui fabrique un objet combinatoire à partir de deux objets donnés. Voilà quelques exemples d'utilisation de cette grammaire et des méthodes `count` et `list` que vous devrez implanter.

```
init_grammar(Perm2)
assert Perm2["Perms"].count(0) == 0 # nb d'objets de taille 0
assert Perm2["Perms"].count(1) == 0 # nb d'objets de taille 1
assert Perm2["Perms"].count(2) == 2 # nb d'objets de taille 2

# liste des objets de taille 0
assert Perm2["Perms"].list([]) == []
# liste des objets étiquetés par un ensemble donné
```

```
assert Perm2["Perms"].list(["a"]) == []
assert Perm2["Perms"].list(["b"]) == []
assert Perm2["Perms"].list(["a","b"]) == [["a","b"], ["b","a"]]
assert Perm2["Perms"].list([1,2]) == [[1,2], [2,1]]
```

1.3.4 Les permutations de toute taille

Voilà le premier exemple de grammaire intéressante, c'est-à-dire qui engendre une famille d'objet quelle que soit la taille. Cela nécessite une définition récursive.

Perms	→	Union(**Empty**, **NonEmpty**)
NonEmpty	→	Product(**Letter**, **Perms**)
Empty	→	Epsilon(permutation vide)
Letter	→	Singleton

En python, on l'implantera de cette façon :

```
Perms = {
    "Perms": UnionRule("Empty","NonEmpty"),
    "NonEmpty": ProductRule("Letter","Perms",lambda l1,l2:l1+l2),
    "Letter": SingletonRule(lambda x:[x]),
    "Empty": EpsilonRule([])
}

init_grammar(Perms)
assert Perms["Perms"].count(0) == 1 # nb d'objets de taille 0
assert Perms["Perms"].count(1) == 1 # nb d'objets de taille 1
assert Perms["Perms"].count(2) == 2 # nb d'objets de taille 2
assert Perms["Perms"].count(3) == 6 # nb d'objets de taille 3
assert Perms["Perms"].count(4) == 24 # nb d'objets de taille 4

# liste des objets de taille 0
assert Perms["Perms"].list([]) == [[]]
# liste des objets étiquetés par un ensemble donné
assert Perms["Perms"].list(["a"]) == [["a"]]
assert Perms["Perms"].list(["b"]) == [["b"]]
assert Perms["Perms"].list(["a","b"]) == [["a","b"], ["b","a"]]
assert Perms["Perms"].list([1,2]) == [[1,2], [2,1]]
assert Perms["Perms"].list([1,2,3]) == [
[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
```

1.3.5 Arbres binaires

Voici pour commencer la grammaire qui engendre les arbres binaires dont les **feuilles** sont étiquetées.

Tree → Union(**Node**, **Leaf**)
Node → Product(**Tree**, **Tree**)
Leaf → Singleton

Ce qui en python, s'implante par

```
TreeLabelLeaves = {
    "Tree": UnionRule("Node", "Leaf"),
    "Node" : ProductRule("Tree", "Tree", Node),
    "Leaf" : SingletonRule(Leaf)
}
```

Remarquez que les objets `Singleton` et `ProductRule` prennent directement les fonctions `Leaf` et `Node` en paramètre pour la création des objets. Voilà les résultats attendus. Notez que la taille de l'arbre est ici le nombre de feuilles.

```
assert TreeLabelLeaves["Tree"].count(0) == 0
assert TreeLabelLeaves["Tree"].count(1) == 1
assert TreeLabelLeaves["Tree"].count(2) == 2
assert TreeLabelLeaves["Tree"].count(3) == 12 # 2 * 6
assert TreeLabelLeaves["Tree"].count(4) == 120 # 5 * 24

assert TreeLabelLeaves["Tree"].list(["a","b"]) == [
Node(Leaf("a"),Leaf("b")), Node(Leaf("b"),Leaf("a"))]
assert len(TreeLabelLeaves["Tree"].list(["a","b","c"])) == 12
assert len(TreeLabelLeaves["Tree"].list(["a","b","c","d"])) == 120
```

Voici l'implantation d'une grammaire pour implanter les arbres dont les noeuds internes sont étiquetés. Cette fois la taille de l'arbre est donnée par le nombre de noeuds internes et les feuilles sont dont de taille 0.

```
TreeLabelNodes = {
    "Tree" : UnionRule("Node", "Leaf"),
    "Node" : ProductRule("Label","Subtrees", lambda l,t: Node(t[0],t[1],l)),
    "Label" : SingletonRule(lambda x:x),
    "Subtrees" : ProductRule("Tree","Tree", lambda t1,t2: (t1,t2)),
    "Leaf" : EpsilonRule(Leaf())
}

init_grammar(TreeLabelNodes)

assert TreeLabelNodes["Tree"].count(0) == 1
assert TreeLabelNodes["Tree"].count(1) == 1
assert TreeLabelNodes["Tree"].count(2) == 4 # 2 * 2
assert TreeLabelNodes["Tree"].count(3) == 30 # 5 * 6
assert TreeLabelNodes["Tree"].count(4) == 336 # 14 * 24

assert len(TreeLabelNodes["Tree"].list(["a","b","c"])) == 30
```

1.4 Les différents produits

Pour que l'on obtienne un comportement intéressant avec les étiquettes, il faut pouvoir contrôler la leurs provenance. On va donc distinguer trois produits.

1. le **produit cartésien ordinaire** noté `Product`. Les étiquettes se répartissent alors de toutes les manières possibles. Ainsi, la règle $\mathbf{C} \to \text{Product}(\mathbf{A}, \mathbf{B})$ sur les étiquettes $[1, 2, 3]$ engendre l'union des ensembles suivant

$$\begin{aligned}
&\mathbf{A}([]) \times \mathbf{B}([1, 2, 3]) \\
&\mathbf{A}([1]) \times \mathbf{B}([2, 3]) \\
&\mathbf{A}([2]) \times \mathbf{B}([1, 3]) \\
&\mathbf{A}([3]) \times \mathbf{B}([1, 2]) \\
&\mathbf{A}([1, 2]) \times \mathbf{B}([3]) \\
&\mathbf{A}([1, 3]) \times \mathbf{B}([2]) \\
&\mathbf{A}([2, 3]) \times \mathbf{B}([1]) \\
&\mathbf{A}([1, 2, 3]) \times \mathbf{B}([])
\end{aligned}$$

2. le **produit cartésien boxed** noté `BoxProduct`. L'ensemble gauche contient forcément la plus petite étiquette. Ainsi, la règle $\mathbf{C} \to \text{BoxProduct}(\mathbf{A}, \mathbf{B})$ sur les étiquettes $[1, 2, 3]$ engendre l'union des ensembles suivant

$$\begin{aligned}
&\mathbf{A}([1]) \times \mathbf{B}([2, 3]) \\
&\mathbf{A}([1, 2]) \times \mathbf{B}([3]) \\
&\mathbf{A}([1, 3]) \times \mathbf{B}([2]) \\
&\mathbf{A}([1, 2, 3]) \times \mathbf{B}([])
\end{aligned}$$

3. le **produit cartésien ordonné** noté `OrdProduct`. Les étiquettes se répartissent en mettant les plus petites à gauches et les plus grandes à droite. Ainsi, la règle $\mathbf{C} \to \text{OrdProduct}(\mathbf{A}, \mathbf{B})$ sur les étiquettes $[1, 2, 3]$ engendre l'union des ensembles suivant

$$\begin{aligned}
&\mathbf{A}([]) \times \mathbf{B}([1, 2, 3]) \\
&\mathbf{A}([1]) \times \mathbf{B}([2, 3]) \\
&\mathbf{A}([1, 2]) \times \mathbf{B}([3]) \\
&\mathbf{A}([1, 2, 3]) \times \mathbf{B}([])
\end{aligned}$$

Avec ces définitions, on peut par exemple définir l'ensemble `SortedSeq` des séquences ordonnées (un seul objet par taille : la séquence ordonnée des étiquettes).

```
SortedSeq = {
    "Sorted": UnionRule("Empty","NonEmpty"),
    "NonEmpty": OrdProdRule("Letter","Sorted",lambda l1,l2:l1+l2),
    "Letter": SingletonRule(lambda x:[x]),
    "Empty": EpsilonRule([])
}
```

```
init_grammar(SortedSeq)
assert SortedSeq["Sorted"].count(0) == 1 # nb d'objets de taille 0
assert SortedSeq["Sorted"].count(1) == 1 # nb d'objets de taille 1
assert SortedSeq["Sorted"].count(2) == 1 # nb d'objets de taille 2
assert SortedSeq["Sorted"].count(3) == 1 # nb d'objets de taille 3
assert SortedSeq["Sorted"].count(4) == 1 # nb d'objets de taille 4

# liste des objets de taille 0
assert SortedSeq["Sorted"].list([]) == [[]]
# liste des objets étiquetés par un ensemble donné
assert SortedSeq["Sorted"].list(["a"]) == [["a"]]
assert SortedSeq["Sorted"].list(["b"]) == [["b"]]
assert SortedSeq["Sorted"].list(["a","b"]) == [["a","b"]]
assert SortedSeq["Sorted"].list([1,2]) == [[1,2]]
assert SortedSeq["Sorted"].list([1,2,3]) == [[1,2,3]]
```

Et voici un exemple utilisant le `BoxProduct`.

```
ExBoxProduct = {
    "BoxProduct": BoxProdRule("Sorted", "Sorted", lambda a,b: (a,b)),
    "Sorted": UnionRule("Empty","NonEmpty"),
    "NonEmpty": OrdProdRule("Letter","Sorted",lambda l1,l2:l1+l2),
    "Letter": SingletonRule(lambda x:[x]),
    "Empty": EpsilonRule([])
}

init_grammar(ExBoxProduct)
assert ExBoxProduct["BoxProduct"].count(0) == 0 # nb d'objets de taille 0
assert ExBoxProduct["BoxProduct"].count(1) == 1 # nb d'objets de taille 1
assert ExBoxProduct["BoxProduct"].count(2) == 2 # nb d'objets de taille 2
assert ExBoxProduct["BoxProduct"].count(3) == 4 # nb d'objets de taille 3

# liste des objets de taille 0
assert ExBoxProduct["BoxProduct"].list([]) == []
# liste des objets étiquetés par un ensemble donné
assert ExBoxProduct["BoxProduct"].list(["a"]) == [(["a"],[])]
assert ExBoxProduct["BoxProduct"].list([1,2]) == [
([1],[2]), ([1,2],[])]
assert ExBoxProduct["BoxProduct"].list([1,2,3]) == [
([1],[2,3]), ([1,2], [3]), ([1,3], [2]), ([1,2,3],[])]
```

1.5 Exercices

Donnez les grammaires engendrant les ensembles suivants. Vous donnerez les descriptions formelles dans le rapports et proposerez aussi des implantations python une fois le projet

réalisé et testerez les différentes méthodes.

1. (niveau 1) La grammaire des **séquences cycliques** décrites en Section 1.1.1. La méthode `count` sur les entiers de 0 à 5 doit vous donner **1, 1, 1, 2, 6, 24**.
2. (niveau 1) La grammaire des arbres binaires dont les feuilles sont étiquetées dans l'ordre de gauche à droite. La méthode `count` sur les entiers de 0 à 5 doit vous donner **0, 1, 1, 2, 5, 14**.
3. (niveau 2) La grammaire des **arbres binaires croissants** décrits en Section 1.1.3. La méthode `count` sur les entiers de 0 à 5 doit vous donner **1, 1, 2, 6, 24, 120**.
4. (niveau 2) La grammaire des **arbres binaires de recherche** décrits en Section 1.1.3. La méthode `count` sur les entiers de 0 à 5 doit vous donner **1, 1, 2, 5, 14, 42**.
5. (niveau 3) La grammaire des **partitions d'ensemble** décrites en Section 1.1.2. La méthode `count` sur les entiers de 0 à 5 doit vous donner **1, 1, 2, 5, 15, 52**.
6. (niveau 3) La grammaire des **Séquences zig-zag** décrites en Section 1.1.1. La méthode `count` sur les entiers de 0 à 5 doit vous donner **1, 1, 1, 2, 5, 16**.

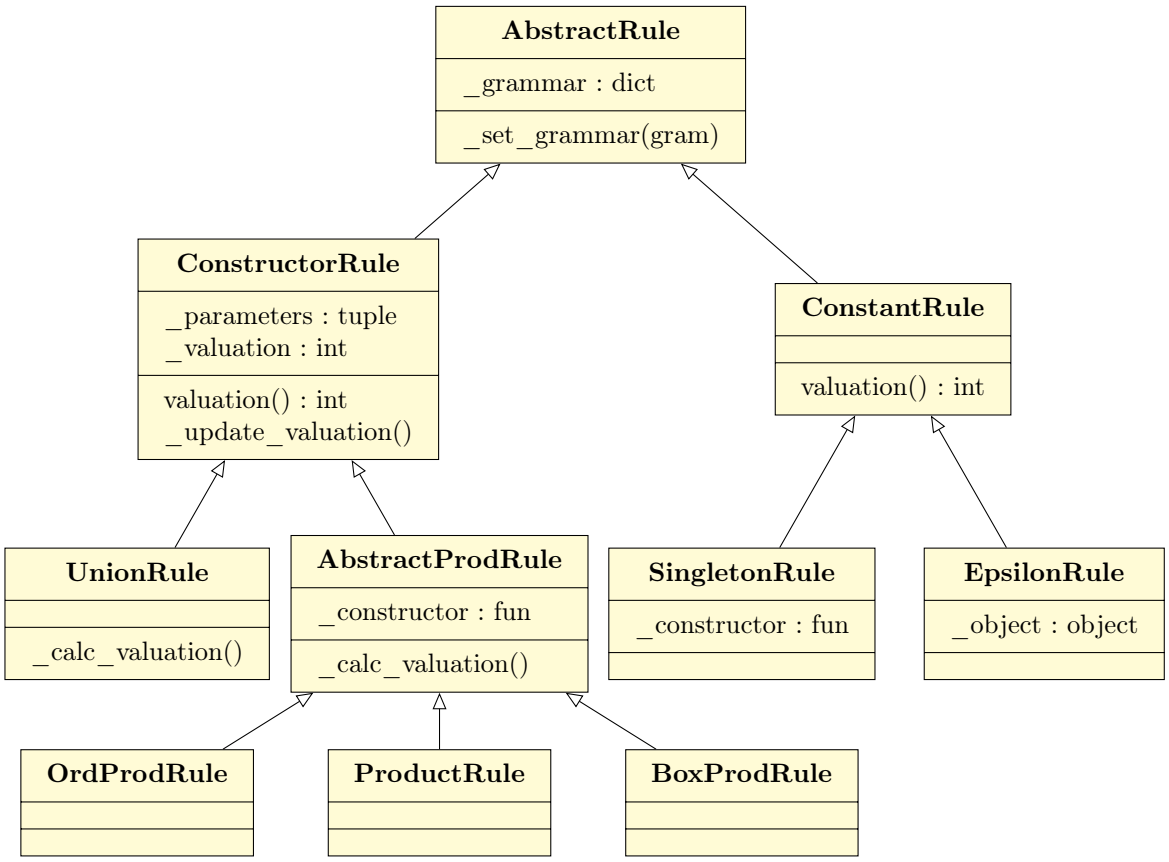
2 Implantation du projet

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 `TreeLabelLeaves["Tree"]` modélise l'ensemble de tous les arbres binaires avec feuille étiquetées (l'équivalent de la classe `BinaryTrees` du TP). Cet ensemble est construit comme l'union de deux sous ensembles modélisés par les objets `TreeLabelLeaves["Node"]` et `TreeLabelLeaves["Leaf"]`. La classe de l'objet `TreeLabelLeaves["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`. On utilisera aussi les classes abstraites `ConstructorRule`, `ConstantRule` et `AbstractProductRule` pour permettre la factorisation du code. (Remarque : en python, rien ne distingue dans la déclaration une classe abstraite d'une classe concrète).

Voici un schéma de la hiérarchie de classe (les méthodes et paramètres des classes sont des suggestions que vous pouvez suivre ou non) :



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`,...

Un fichier `projet.py` vous est fourni avec l'architecture de base. Vous devez conservez les noms de classe ainsi que les constructeurs des classes concrètes (`UnionRule`, `OrdProdRule`, `ProductRule`, `BoxProdRule`, `SingletonRule`, `EpsilonRule`) pour que les tests définis dans `tests.py` puissent être exécutés. En dehors de ça, vous êtes libre de modifier ce que vous souhaitez et de rajouter toutes les méthodes dont vous avez besoin. Si vous travaillez sur Jupyter, vous pouvez importer ou copier / coller le code du fichier dans une cellule.

2.2 Implantation de la définition récursive

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.

Implantez une méthode `set_grammar` dans la classe `AbstractRule` et implantez les fonctions `save_grammar` et `check_grammar`..

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. Nous les avons 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 quatre 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 des non-terminaux N_1 et N_2 est la somme des valuations de N_1 et de N_2 ; Par ailleurs, la valuation du `BoxProduct` est au minimum de 1.

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 avec feuilles étiquetées, 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 :

1. À rendre sur papier : Pour les différentes grammaire considérée (exemples et exercices), appliquer l'algorithme à la main et donner les valuations des différents non terminaux.
2. 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 :

3. 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

On applique récursivement les définitions des constructeurs `Singleton`, `Epsilon`, `Union` et `Prod` pour construire la liste des objets de taille i.

3.3.1 Singleton et Epsilon

Dans le cas des règles constantes, on vérifiera si la taille de la liste des étiquettes correspond à l'unique taille engendrée par la règle (1 pour Singleton, 0 pour Epsilon) et le cas échéant, on engendra cet objet unique. Pour Singleton, il faudra utiliser la fonction de création de l'objet en fonction de l'étiquette. Pour Epsilon, l'objet est stocké dans la classe.

3.3.2 Union

Pour une règle union de deux règles R_1 et R_2, on engendrera récursivement les objets de R_1 puis ceux de R_2 sur l'ensemble des étiquettes données en paramètre.

3.3.3 Produit

La difficulté principale vient des différents produits car on doit dans ce cas décider en plus de la répartition des étiquettes.

Pour chaque produit, on écrira une méthode `iter_label` qui prend en paramètre un entier k et une liste de labels ℓ et qui itère selon toutes les façons de diviser la liste ℓ en k éléments d'un côté et $n-k$ éléments de l'autre selon les règles particulières du produit décrite en Section 1.4.

Exemples :

- la méthode `iter_label([1,2,3],2)` sur le produit classique doit engendrer les découpages suivants : $([1,2],[3]),([1,3],[2]),([2,3],[1])$.
- la méthode `iter_label([1,2,3],2)` sur le produit box doit engendrer les découpages suivants : $([1,2],[3]),([1,3],[2])$.
- la méthode `iter_label([1,2,3],2)` sur le produit ordonné doit engendrer l'unique découpage : $([1,2],[3])$.

L'algorithme de création des objets est ensuite le même pour tous les produits. Pour calculer le produit de deux ensembles E_1 et E_2 : pour toutes les valeurs de k en fonction des valuations de E_1 et E_2, pour tous les découpages possible de `iter_label`, pour tous les objets e_1 de E_1 de taille k et pour tous les objets e_2 de E_2 de taille $n-k$, on créera l'objet formé de e_1 et e_2 grâce à la fonction de création stockée dans la classe.

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 dont les feuilles sont étiquetées par $[1,2,3,4,5,6]$ de rang 12. 12.

```
>>> TreeLabelLeaves["Tree"].unrank([1,2,3,4,5,6],12)
(1, (2, (4, (3, (5, 6)))))
```

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.
- ToDo: UPDATE 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`.

4. Écrire, dans le rapport, la spécification la plus précise possible que doivent vérifier les différentes méthode d'une classe combinatoire.
5. 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.
6. 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 les différentes règles de produit, il faut une fonction permettant de retrouver le couple `(a, b)` à partir de `obj`.

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

4.3 Caching

8. 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

9. 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.

10. 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`.

11. 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))
```

Bon Travail !