Université Paris-Saclay

Master 2 FIIL

Comptage et Énumération

Projet

Génération d'objets combinatoires décrits par une grammaire

Fait par :
Bryce Tichit

Professeurs :
Florent Hivert
Viviane Pons

26 novembre 2017

Table des matières

Questions de cours

Question 1

Pour les grammaires des arbres et des mots de Fibonnacci, donner dans un tableau pour n = 0, . . . , 10 les réponses attendue pour la méthode count pour les 8 non terminaux des mots de Fibonnacci et les 3 non terminaux des arbres binaires

---

	0	1	2	3	4	5	6	7	8	9	10
Fib	1	2	3	5	8	13	21	34	55	89	144
Cas1	0	2	3	5	8	13	21	34	55	89	144
Cas2	0	1	1	2	3	5	8	13	21	34	55
Vide	1	0	0	0	0	0	0	0	0	0	0
CasAu	0	1	2	3	5	8	13	21	34	55	89
AtomA	0	1	0	0	0	0	0	0	0	0	0
AtomB	0	1	0	0	0	0	0	0	0	0	0
CasBAu	0	0	1	2	3	5	8	13	21	34	55

	0	1	2	3	4	5	6	7	8	9	10
Tree	0	1	1	2	5	14	42	132	429	1430	4862
Node	0	0	1	2	5	14	42	132	429	1430	4862
Leaf	0	1	0	0	0	0	0	0	0	0	0

Question 2

Donner la grammaire de tous les mots sur l'alphabet A,B.

```
allABwords = {
    "Axiom": UnionRule("Axiom2","vide"),
    "Axiom2": ProductRule("Axiom","letter",lambda a,b: a+b),
    "vide": EpsilonRule(""),
    "letter": UnionRule("a","b"),
    "a": SingletonRule("a"),
    "b": SingletonRule("b")
}
init_grammar(allABwords)
```

```
Valuation = {'Axiom': 0, 'Axiom2': 1, 'vide': 0, 'letter': 1, 'a': 1, 'b': 1}
```

Question 3

Donner la grammaire des mots de Dyck, c'est-à-dire les mots sur l'alphabet {(,)} et qui sont correctement parenthésés.

```
dyckGrammar = {
    "Axiom": UnionRule("dyck","enddyck"),
    "dyck": ProductRule("dyck2","Axiom",lambda a,b: a+b),
    "dyck2" : ProductRule("parleft","intermdyck",lambda a,b: a+b),
    "enddyck" : EpsilonRule(""),
    "parleft" : SingletonRule("("),
```

```
    "parright" : SingletonRule(")"),
    "intermdyck": ProductRule("Axiom","parright",lambda a,b: a+b)
    }
init_grammar(dyckGrammar)
```

```
Valuation =
{'Axiom': 0, 'dyck': 2, 'dyck2': 2, 'enddyck': 0, 'parleft': 1,
'parright': 1, 'intermdyck': 1}
```

Question 4

Donner la grammaire de mots sur l'alphabet A,B qui n'ont pas trois lettres consécutivement égales.

```
No3ConsecutiveGram = {
"Vide" : EpsilonRule("") ,
"A" : SingletonRule("A"),
"B" : SingletonRule("B"),
"S" : UnionRule("Vide","2letterleft"),
"2letterleft" : UnionRule("a_2aleft","b_2bleft"),

"a_2aleft" : ProductRule("A", "1aleft",lambda a,b: a+b),
"b_2bleft" : ProductRule("B", "1bleft",lambda a,b: a+b),

"a_1aleft" : ProductRule("A", "b_2bleft",lambda a,b: a+b),
"b_1aleft" : ProductRule("B","1bleft",lambda a,b: a+b),
"1aleft" : UnionRule("a_1aleft","eps_1aleft"),
"eps_1aleft" : UnionRule("b_1aleft","Vide"),

"a_1bleft" : ProductRule("A", "1aleft",lambda a,b: a+b),
"b_1bleft" : ProductRule("B","a_2aleft",lambda a,b: a+b),
"1bleft" : UnionRule("a_1bleft","eps_1bleft"),
"eps_1bleft" : UnionRule("b_1bleft","Vide")
```

```
}
init_grammar(No3ConsecutiveGram)

Valuation =
{'Vide': 0, 'A': 1, 'B': 1, 'S': 0, '2letterleft': 1, 'a_2aleft': 1, 'b_2bleft': 1,
'a_1aleft': 2, 'b_1aleft': 1, '1aleft': 0,
'eps_1aleft': 0, 'a_1bleft': 1, 'b_1bleft': 2, '1bleft': 0, 'eps_1bleft': 0}
```

Question 5

Donner la grammaire des palindromes sur l'alphabet A, B, même question sur l'alphabet A,B,C.

```
palinABGram = {
 "A" : SingletonRule("A"),
 "B" : SingletonRule("B"),
 "Vide" : EpsilonRule(''),
 "Axiom" : UnionRule("PalinA", "Palin_e_b"),
 "Palin_e_b" : UnionRule("PalinB","Vide"),
 "PalinA" : ProductRule("A","PalinA_suite",lambda a,b: a+b),
 "PalinA_suite" : ProductRule("Axiom","A",lambda a,b: a+b),
 "PalinB" : ProductRule("B","PalinB_suite",lambda a,b: a+b),
 "PalinB_suite" : ProductRule("Axiom","B",lambda a,b: a+b)
}
init_grammar(palinABGram)

Valuation = {'A': 1, 'B': 1, 'Vide': 0, 'Axiom': 0, 'Palin_e_b': 0,
'PalinA': 2, 'PalinA_suite': 1, 'PalinB': 2, 'PalinB_suite': 1}

palinABCGram = {
 "A" : SingletonRule("A"),
 "B" : SingletonRule("B"),
 "C" : SingletonRule("C"),
 "Vide" : EpsilonRule(''),
```

```
 "Axiom" : UnionRule("PalinA", "Palin_e_b"),
 "Palin_e_b" : UnionRule("PalinB_C","Vide"),
 "PalinB_C" : UnionRule("PalinB", "PalinC"),
 "PalinA" : ProductRule("A","PalinA_suite",lambda a,b: a+b),
 "PalinA_suite" : ProductRule("Axiom","A",lambda a,b: a+b),
 "PalinB" : ProductRule("B","PalinB_suite",lambda a,b: a+b),
 "PalinB_suite" : ProductRule("Axiom","B",lambda a,b: a+b),
 "PalinC" : ProductRule("C","PalinC_suite",lambda a,b: a+b),
 "PalinC_suite" : ProductRule("Axiom","C",lambda a,b: a+b)
}
init_grammar(palinABCGram)

Valuation =
{'A': 1, 'B': 1, 'C': 1, 'Vide': 0, 'Axiom': 0, 'Palin_e_b': 0, 'PalinB_C': 2,
'PalinA': 2, 'PalinA_suite': 1, 'PalinB': 2, 'PalinB_suite': 1, 'PalinC': 2, 'PalinC_
```

Question 6

Donner la grammaire des mots sur l'alphabet A,B qui contiennent autant de lettres A que de lettres B.

```
autantABGram = {
"lettre_A" : SingletonRule("A"),
"lettre_B" : SingletonRule("B"),
"Axiom" : UnionRule("aB", "bA"),

"aB" : ProductRule("lettre_A", "B", lambda a,b: a+b),
"bA" : ProductRule("lettre_B","A", lambda a,b: a+b),
"A" : UnionRule("lettre_A", "A_suite"),
"A_suite" : UnionRule("aS", "bAA"),
"aS" : ProductRule("lettre_A","Axiom",lambda a,b: a+b),
"bAA" : ProductRule("lettre_B", "AA", lambda a,b: a+b),
"AA" : ProductRule("A","A",lambda a,b: a+b),

"bA" : ProductRule("lettre_B", "A", lambda a,b: a+b),
```

```
"aB" : ProductRule("lettre_A","B", lambda a,b: a+b),
"B" : UnionRule("lettre_B", "B_suite"),
"B_suite" : UnionRule("bS", "aBB"),
"bS" : ProductRule("lettre_B","Axiom",lambda a,b: a+b),
"aBB" : ProductRule("lettre_A", "BB", lambda a,b: a+b),
"BB" : ProductRule("B","B",lambda a,b: a+b)
}
init_grammar(autantABGram)

Valuation = {'lettre_A': 1, 'lettre_B': 1, 'Axiom': 2, 'aB': 2, 'bA': 2, 'A': 1, 'A_s
'aS': 3, 'bAA': 3, 'AA': 2, 'B': 1, 'B_suite': 3, 'bS': 3, 'aBB': 3, 'BB': 2}
```

Question 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.

On effectue cette vérification dans la fonction init_ grammar,

```
In [ ]: def init_grammar(gram):

            for ruleName, ruleDef in gram.items():

                if not ruleDef._constant:
                    for pr in ruleDef._parameters :
                        if pr not in gram:
                            raise ValueError("
                  A rule of the grammar ("+pr+") does not exist")

                ...
```

1 Implémentation et algorithmes

1.1 Implémentation des classes de grammaire

Nous commençons par implémenter les différentes classes qui modéliseront la grammaire en `Python`, on suivra la hiérarchie suivante :

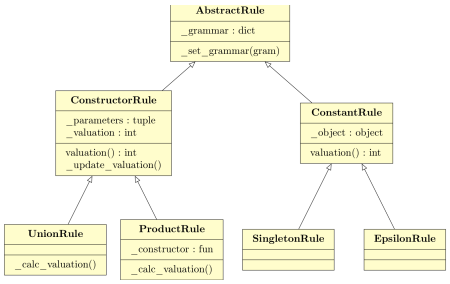

FIGURE 1 – La hiérarchie des classes grammaire

— **AbstractRule** : Représente une abstraction des règles de la grammaire, en particulier elle implémentera un dictionnaire qui fera référence à la grammaire toute entière ainsi qu'une fonction **set_grammar** qui permettra de modifier la grammaire (?)
— **ConstructorRule** : Règles de constructions qui répresentent les symboles non-terminaux et qui sont dans notre cas **UnionRule** et **ProductRule**
  — ProductRule : Cette règle représente le produit cartésien de deux règles, elle hérite de la classe ConstructorRule
  — UnionRule : Il s'agit de l'union disjointe de deux règles, elle hérite également de la classe ConstructorRule
— **ConstantRule** : Règles constantes représentant les symboles terminaux
  — EpsilonRule : Règle représentant le symbole ϵ
  — SingletonRule : Règle représentant un symbole terminal quelconque

On implémente la classe abstraite avec la metaclass *ABCMeta* et les fonctions abstraites sont décorées avec *@abstractmethod* afin de respecter la hiérarchie.

On rajoute l'attribut `_constant` dans la classe *AbstractRule* afin de savoir si la règle concrète est constante ou non.

On définit également une méthode `self.rules` pour la classe *ConstructorRule* qui renvoie les deux règles qui la compose sous forme de couple. On décore cette mé-

thode avec `@property` afin de pouvoir l'appeler sans parenthèses (simple question d'esthétique).

1.2 Comptage : Méthode count

La méthode *count* prends une règle de grammaire et un entier n afin de renvoyer le nombre d'éléments générables par la règle de grammaire de taille exactement n.

Dans le cas d'une *UnionRule* on somme simplement le nombre d'élements des 2 règles,

$$count_{unionRule}(rule_1, rule_2, n) = count(rule_1, n) + count(rule_2, n)$$

Dans le cas d'une *ProductRule* il faut compter le nombre d'éléments pour chaque possibilité de répartition de n dans les deux règles, c'est à dire pour tous les $i, k \geqslant 0$ tel que $i + k = n$. On obtient le résultat suivant,

$$count_{productrule}(rule_1, rule_2, n) = \sum_{i+k=n} count(rule_1, i) \cdot count(rule_2, k)$$

On implémente cela avec une boucle `for` en `Python`, avec cependant une optimisation. En effet si la valuation de $rule_1$ est plus grande que i alors il n'existe aucun objet de taille i dans $rule_1$ ce qui fait que nous pouvons passer ce calcul qui ferait des appels récursifs inutiles, pareil si la valuation de $rule_2$ est plus grande que k.

Les cas *EpsilonRule* et *SingletonRule* sont triviaux, une règle *EpsilonRule* contient un seul élément de taille 0 et une règle *SingletonRule* contient un seul élément qui est de taille 1. On a donc,

$$count_{singletonRule}(rule, 1) = 1$$
$$count_{singletonRule}(rule, n) = 0$$

$$count_{epsilonRule}(rule, 0) = 1$$
$$count_{epsilonRule}(rule, n) = 0$$

1.3 Énumération : Méthode list

La méthode *list* prends une règle de grammaire et un entier n afin de renvoyer la liste des éléments générables par la grammaire de taille exactement n.

Dans le cas d'une *UnionRule* on concatène simplement la liste d'élements des 2 règles,

$$list_{unionRule}(rule_1, rule_2, n) = list(rule_1, n) \cup list(rule_2, n)$$

Dans le cas d'une *ProductRule* on suit la même logique que pour $count_{productRule}$, on énumère chaque éléments générable par la grammaire en prenant compte de chaque possibilité pour i et k. L'implémentation en `Python` est la suivante,

```
def list(self, n):
                rule1, rule2 = self.rules
                res=[]

                for i in range(n+1):
                    k=n-i
                    if rule1.valuation > i or rule2.valuation > k:
                        continue
                    for el1 in rule1.list(i):
                        for el2 in rule2.list(k):
                            res.append(self._cons(el1,el2))
                return res
```

On implémente également la même optimisation que pour count quant à la valuation des deux règles. Ainsi on énumère chaque possibilité, qu'on ajoute à la liste de retour en prenant bien soin de lui appliquer la bijection `self._cons` au préalable.

Comme précedemment, les cas *EpsilonRule* et *SingletonRule* sont triviaux. On renvoie le singleton contenant l'élément associé à la règle dans le cas où $n = 1$ pour *SingletonRule* et dans le cas où $n = 0$ pour *EpsilonRule*. Dans tout les autres cas on renvoie la liste vide.

1.4 Élément d'indice donné : Méthode unrank

La méthode *unrank* prends une règle de grammaire, un entier n et un entier r afin de renvoyer l'élement d'indice r dans la liste des éléments générable par la grammaire de taille n.

L'idée pour l'implémentation de cette méthode sur une *UnionRule* est la suivante, sachant que cette règle est composée de deux règles $rule_1$ et $rule_2$ alors dans la liste des éléments de la grammaire on aura d'abord tous les éléments générables par $rule_1$

puis tous les éléments générables par $rule_2$. On regarde ainsi le nombre d'éléments générables par $rule_1$, selon cette valeur nous avons deux cas de figure.

— Soit $r < count(rule_1)$ à ce moment là on renvoie simplement l'élement r de $rule_1$ (l'élement se trouve dans le premier ensemble)

— Soit $r \geqslant count(rule_2)$ et à ce moment là on renvoie l'élement $r - count(rule_2)$ de $rule_2$ (l'élement se trouve dans le second ensemble)

Dans le cas *ProductRule* l'idée est légèrement plus compliquée mais reste similaire à tout ce qui nous avons fait jusqu'a maintenant, en effet il suffit de compter !

Rappelons la formule de count pour *ProductRule*,

$$count_{productrule}(rule_1, rule_2, n) = \sum_{i+k=n} count(rule_1, i) \cdot count(rule_2, k)$$

À chaque itération de notre boucle on compte de plus en plus d'éléments, on compte ainsi jusqu'a dépasser la valeur r qui est l'indice de l'élement souhaité. Supposons que notre boucle s'arrête ainsi à l'itération i, notre élement se trouve donc juste entre les éléments comptés à l'itération précedente et l'itération actuelle (et terminale). En d'autres termes notre élément se trouve dans l'ensemble $rule_1(i-1) \times rule_2(n-i-1)$. Il nous reste à trouver quel est l'indice j de cet élément dans le dît ensemble, celui-ci est simplement la différence entre le rang souhaité et le comptage à l'itération juste avant.

On connait maintenant quel élément de quel ensemble renvoyer, on décompose cet élément comme faisant partie du produit cartésien des deux ensembles $rule_1(i-1)$ et $rule_2(n-i-1)$ et on calcule les rangs des deux parties grâce à la division et au reste de j par le nombre d'éléments dans $rule_2$. On applique donc la bijection à ces deux éléments avant de renvoyer le résultat. L'implémentation en `Python` est la suivante,

```
def unrank(self, n, rank):
                if rank >= self.count(n):
                    raise ValueError("Error called unrank with a too big rank")

                rule1, rule2 = self.rules
                count, prec_count, i = 0, 0, 0
                while count <= rank:
                    k = n - i
                    prec_count=count
                    count += rule1.count(i) * rule2.count(k)
```

```
        i += 1
    i-=1
    j = rank - prec_count
    k = rule2.count(n-i)
    q, r = j // k, j % k

    return self._cons(rule1.unrank(i, q), rule2.unrank(n-i, r))
```

Dans le cas *ProductRule* et *SingletonRule* on se contente de renvoyer les éléments sous réserve que les arguments soient bons. Sinon on lève une exception. On lève également une exception **ValueError** pour les autre cas si on demande un rang supérieur à la taille de l'ensemble.

2 Améliorations

2.1 Indice d'un élément donné : Méthode Rank

La méthode rank prends une règle de grammaire et un objet généré par cette grammaire afin de renvoyer l'indice r de cet élement dans la liste des éléments générable par la grammaire de la taille de l'objet.

L'idée derrière *rank* est très similaire à *unrank*, il faut cependant que les classes sachent analyser les objets. Pour utiliser rank il faut donner à la classe deux fonctions,

- `objSize`, une fonction prenant un élément et renvoyant sa taille.
- Pour *UnionRule* il faut la fonction `origin`, une fonction renvoyant à quelle règle l'élement issu de l'*UnionRule* appartient.
- Pour *ProductRule* il faut la fonction `uncons`, une fonction permettant de retrouver les deux éléments ayant permis de produire l'objet par le biais de `cons`.

Une fois que nous avons ces deux fonctions l'implémentation est simple et très similaire à *unrank*.

Dans le cas *UnionRule*, nous avons deux possibilités.

- Soit l'objet provient de la première règle, à ce moment là on rapelle rank récursivement sur la première règle.
- Soit l'objet provient de la seconde, à ce moment là on renvoie le nombre d'éléments de $rule_1$ + le rang de l'objet dans $rule_2$

Dans le cas *ProductRule*, notre objet est donc composé de deux sous-objets obj_1 et obj_2. On commence donc par compter le nombre d'élements avec un obj_1 inférieur à celui de nôtre objet.

On rajoute à cela le rang de obj_2 dans sa règle, le produit de rang de obj_1 et du nombre d'objets dans obj_2 de taille size_obj2 afin d'obtenir le rang.

Pour les règles *SingletonRule* on renvoie simplement 0 si l'objet est bien celui représenté par la règle et on lève une exception dans tous les autres cas.

2.2 Caching

Afin d'avoir un programme plus utilisable nous avons ajouté la mémoïzation des méthodes combinatoires, ceci grâce au décorateur `lru_cache` de la librairie `functools`. Ce dernier implémente une table de hachage afin d'avoir un accès au cache constant et remplace les éléments les plus vieux du cache lorsque celui-ci atteint sa taille maximale (least recently used).

Par défaut la taille du cache est de 50 mais il est possible de le changer. La variable qui contrôle cette taille de cache s'apelle `MAX_CACHE_SIZE`. Elle se trouve au début de notre programme.

Ce cache permet un gain de performances énorme et permet d'obtenir un compromis satisfaisant face au grand nombre d'appels récursifs de nos méthodes.

2.3 Spécification fonctionelle et tests

Afin de d'assurer que notre programme est bien correct nous utilisons des tests unitaires, ceux-ci s'appuient sur la spécification fonctionelle suivante.

- self.unrank(n, k) == self.list(n)[k]
- self.count(n) == len(self.list(n))
- self.unrank(n, k) lève une exception **ValueError** si $k \geqslant n$
- self.rank(self.unrank(x, n)) == x **for** x **in** range(self.count(n)) $\forall n$

On écrit une fonction runTest dans `tests.py` qui s'occupe de lancer l'ensemble des tests sur une règle, on ne peut tester nos fonctions avec l'ensemble des entrées possibles car il y en a un nombre infini. Lorsque ce genre de cas se présente on prends simplement un nombre d'entrées (par défaut 5) aléatoires et on lance les tests sur ceux-ci.

On aurait pu utiliser un framework de test pour Python mais étant donné qu'il n'y avait pas un grand nombre de tests à faire nous n'avons pas jugé ça utile.

2.4 Grammaires expressives

Pour simplifier l'écriture des grammaires nous implémentons les grammaires expressives. On écrit ainsi de nouvelles classes pour cela, on peut par la suite "augmenter" la grammaire avec la fonction `expand_init_grammar`.

```
In [12]: condensedTreeGram = {"Tree" :
Union (Singleton(Leaf),
Prod(NonTerm("Tree"), NonTerm("Tree"), "".join))}
```

```
In [13]: expandedTreeGram = expand_init_grammar(condensedTreeGram)
         expandedTreeGram
```

```
Out[13]: {'Prod_Tree_Tree': ProductRule(Tree,Tree),
          'Sing_leaf': SingletonRule(leaf),
          'Tree': UnionRule(Sing_leaf,Prod_Tree_Tree)}
```

```
In [14]: treeGram
```

```
Out[14]: {'Leaf': SingletonRule(leaf),
          'Node': ProductRule(Tree,Tree),
          'Tree': UnionRule(Node,Leaf)}
```

2.5 Règle BoundRule

On implémente la règle **BoundRule** comme une classe héritère de *ConstructorRule*, cette règle prends une *ConstructorRule* et deux bornes afin de représenter l'ensemble des éléments générables par la règle *ConstructorRule* dont la taille est compris entre les deux bornes (la borne supérieure est incluse).

Le paramètre n correspondant est ignoré par cette règle lorsqu'on l'apelle avec `list` ou `count`.

```
In [15]: from examplegrammars import dyckGram

         dg=dyckGram["Axiom"]
         BoundRule(dg,0,8).list(0)
```

```
Out[15]: ['',
          '()',
          '()()',
          '(())',
          '()()()',
          '()(())',
          '(())()',
          '(()())',
          '((()))',
          '()()()()',
            ...
```

2.6 Règle SequenceRule

On peut utiliser la règle **SequenceRule**, celle-ci est implémentée comme héritère de *ConstructorRule*.

La syntaxe est SequenceRule(nonTerm,nonTermVide,cons).

```
In [16]: sizestr = lambda w: len(w)
         testSequence = {
             'letter_a' : SingletonRule('a'),
             'vide' : EpsilonRule(''),
             'sequence' : SequenceRule("letter_a", "vide",
             lambda a,b: a+b, lambda w: (w[0],w[1:]), sizestr)
         }
         init_grammar(testSequence)
         BoundRule(testSequence["sequence"],0,10).list(0)

Out[16]: ['',
          'a',
          'aa',
          'aaa',
          'aaaa',
          'aaaaa',
          'aaaaaa',
          'aaaaaaa',
          'aaaaaaaa',
```

```
'aaaaaaaaa',
'aaaaaaaaaa']
```