Rapport Projet Combinatoire

Samia Nabili
Travail réalisé en binôme avec Viktor Frelikh

24 novembre 2017

1 Réponses attendues pour la méthode count

Pour la grammaire des arbres :

n	0	1	2	3	4	5	6	7	8	9	10
count(n, Tree)	0	1	1	2	5	14	42	123	429	1430	4862
count(n, Node)	0	0	1	2	5	14	42	123	429	1430	4862
count(n, Leaf)	0	1	0	0	0	0	0	0	0	0	0

Pour la grammaire des mots de Fibonacci :

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

2 Grammaires

2.1 Alphabet A,B

$S = \mathcal{E} \mid AS \mid BS$
w est un mot de la grammaire A, B si :
— soit w est vide
— soit w est de la forme Au où u est un mot de la grammaire A, B
— soit w est de la forme Bu où u est un mot de la grammaire A, B

2.2 Mots de Dyck

$S = \mathcal{E} \mid (S) \mid S(S)$
w est un mot de Dyck si :
— soit w est vide
— soit w est de la forme (u) où u est un mot de Dyck de Dyck
— soit w est de la forme $u(v)$ où u et v sont des mots de Dyck

2.3 Mots sur l'alphabet A,B qui n'ont pas trois lettres consécutives égales

$S = \mathcal{E} \mid U \mid T$
$U = A \mid AA \mid AT \mid AAT$
$T = B \mid BB \mid BT \mid BBT$
w est un mot de cette grammaire si :
— soit w est vide
— soit w est de la forme A, AA, B, BB
— soit w est de la forme AT ou AAT où T est un mot de la grammaire qui commence par B
— soit w est de la forme BU ou BBU où U est un mot de la grammaire qui commence par A

2.4 Palindromes sur l'alphabet A, B

$S = \mathcal{E} \mid A \mid B \mid ASA \mid BSB$
w est un mot de la grammaire des palindrome sur A, B si :
— soit w est vide
— soit w est de la forme A ou B
— soit w est de la forme AuA où u est un palindrome.
— soit w est de la forme BuB où u est un palindrome.

2.5 Palindromes sur l'alphabet A, B, C

$S = \mathcal{E} \mid A \mid B \mid ASA \mid BSB \mid CSC$
w est un mot de la grammaire des palindrome sur A, B si :
— soit w est vide
— soit w est de la forme A ou B ou C
— soit w est de la forme AuA où u est un palindrome.
— soit w est de la forme BuB où u est un palindrome.
— soit w est de la forme CuC où u est un palindrome.

2.6 Mots sur l'alphabet A,B qui contiennent autant de A que de B

$S = \mathcal{E} \mid aTbS \mid bUaS$
$T = \mathcal{E} \mid aTbT$
$U = \mathcal{E} \mid bTaT$

3 Calcul de la valuation

3.1 Mots de Fibonacci

n	Fib	Cas1	Cas2	Vide	CasAu	AtomA	AtomB	CasBAu
règle	Vide ∪ Cas1	CasAU ∪ Cas2	AtomA ∪ AtomB	$\mathcal{E}$	AtomA*Fib	A	B	AtomB*CasAu
0	∞	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	∞	0	∞	1	1	∞
2	0	∞	1	0	1	1	1	∞
3	0	1	1	0	1	1	1	2
3	0	1	1	0	1	1	1	2

3.2 Mots de Dyck

n	Dyck	Casuu	Vide	AtomLPAR	AtomRPAR	Cas(u	Casu)
règle	Vide ∪ Casuu	Dyck * Cas(u	E	"("	")"	LPAR * Casu)	Dyck * RPAR
0	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	0	1	1	∞	∞
2	0	∞	0	1	1	∞	∞
3	0	∞	0	1	1	∞	1
4	0	∞	0	1	1	2	1
5	0	2	0	1	1	2	1
6	0	2	0	1	1	2	1

3.3 Mots sur l'alphabet A,B

n	AB	AtomA	AtomB	CasAB	Vide	CasAu	CasBu
règle	Vide ∪ CasAB	A	B	CasAu ∪ CasBu	$\mathcal{E}$	AtomA * AB	AtomB * AB
0	∞	∞	∞	∞	∞	∞	∞
1	∞	1	1	∞	0	∞	∞
2	0	1	1	∞	0	1	1
3	0	1	1	1	0	1	1
4	0	1	1	1	0	1	1

3.4 Mots qui n'ont pas trois lettres consécutives égales sur A, B

n	Three	Vide	AtomA	AtomB	AA	BB	S	U	U1
règle	Vide ∪ S	$\mathcal{E}$	A	B	AtomA*AtomA	AtomB*AtomB	U ∪ T	AtomA ∪ U1	AA ∪ U2
0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	∞	0	1	1	∞	∞	∞	∞	∞
2	0	0	1	1	2	2	∞	1	∞
3	0	0	1	1	2	2	1	1	2
4	0	0	1	1	2	2	1	1	2
5	0	0	1	1	2	2	1	1	2
6	0	0	1	1	2	2	1	1	2

n	U2	AT	AAT	T	T1	T2	BU	BBU
règle	AT ∪ AAT	AtomA * T	AtomA * AT	AtomB ∪ T1	BB ∪ T2	BU ∪ BBU	AtomB * U	AtomB * BU
0	∞	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	1	∞	∞	∞	∞
3	∞	2	∞	1	2	∞	2	∞
4	2	2	3	1	2	∞	2	3
5	2	2	3	1	2	2	2	3
6	2	2	3	1	2	2	2	3

3.5 Palindromes sur A,B

n	Pal	Vide	AtomA	AtomB	S	S1
règle	Vide ∪ S	$\mathcal{E}$	A	B	AtomA ∪ S1	AtomB ∪ S2
0	∞	∞	∞	∞	∞	∞
1	∞	0	1	1	∞	∞
2	0	0	1	1	1	1
3	0	0	1	1	1	1
4	0	0	1	1	1	1
5	0	0	1	1	1	1
6	0	0	1	1	1	1

n	S2	ASA	ASA1	BSB	BSB1
règle	ASA ∪ BSB	AtomA * ASA1	Pal * AtomA	AtomB * BSB1	Pal * AtomB
0	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	∞	∞	1	∞	1
4	∞	2	1	2	1
5	2	2	1	2	1
6	2	2	1	2	1

3.6 Palindromes sur A,B,C

n	Pal	Vide	AtomA	AtomB	AtomC	S	S1	S2	S3
règle	Vide ∪ S	$\mathcal{E}$	A	B	C	AtomA ∪ S1	AtomB ∪ S2	AtomC ∪ S3	ASA ∪ S4
0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	∞	0	1	1	1	∞	∞	∞	∞
2	0	0	1	1	1	1	1	1	∞
3	0	0	1	1	1	1	1	1	∞
4	0	0	1	1	1	1	1	1	∞
5	0	0	1	1	1	1	1	1	2
6	0	0	1	1	1	1	1	1	2

n	S4	ASA	ASA1	BSB	BSB1	CSC	CSC1
règle	BSB ∪ CSC	AtomA * ASA1	Pal * AtomA	AtomB * BSB1	Pal * AtomB	AtomC * CSC1	Pal * AtomC
0	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	1	∞	1	∞	1
4	∞	2	1	2	1	2	1
5	2	2	1	2	1	2	1
6	2	2	1	2	1	2	1

3.7 Palindromes sur A,B,C

n	Pal	Vide	AtomA	AtomB	AtomC	S	S1	S2	S3
règle	Vide ∪ S	$\mathcal{E}$	A	B	C	AtomA ∪ S1	AtomB ∪ S2	AtomC ∪ S3	ASA ∪ S4
0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	∞	0	1	1	1	∞	∞	∞	∞
2	0	0	1	1	1	1	1	1	∞
3	0	0	1	1	1	1	1	1	∞
4	0	0	1	1	1	1	1	1	∞
5	0	0	1	1	1	1	1	1	2
6	0	0	1	1	1	1	1	1	2

n	S4	ASA	ASA1	BSB	BSB1	CSC	CSC1
règle	BSB ∪ CSC	AtomA * ASA1	Pal * AtomA	AtomB * BSB1	Pal * AtomB	AtomC * CSC1	Pal * AtomC
0	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	1	∞	1	∞	1
4	∞	2	1	2	1	2	1
5	2	2	1	2	1	2	1
6	2	2	1	2	1	2	1

3.8 Mots sur A,B qui contiennent autant de A que de B

n	Vide	AtomA	AtomB	S	S1	T	U	aTbS	TbS	bS
règle	$\mathcal{E}$	A	B	$\mathcal{E} \cup$ S1	aTbS $\cup$ bUaS	$\mathcal{E} \cup$ aTbT	$\mathcal{E} \cup$ bUaU	A * TbS	T * bS	B * S
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	1	∞	∞	∞	∞	∞	∞	∞
2	0	1	1	0	∞	0	0	∞	∞	∞
3	0	1	1	0	∞	0	0	∞	∞	1
4	0	1	1	0	∞	0	0	∞	1	1
5	0	1	1	0	∞	0	0	2	1	1
6	0	1	1	0	2	0	0	2	1	1
7	0	1	1	0	2	0	0	2	1	1

n	bUaS	UaS	aS	aTbT	TbT	bT	bUaU	UaU	aU
règle	B * UaS	U * aS	A * S	A * TbT	T * bT	B * T	B * UaU	U * aU	A * U
0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	1	∞	∞	1	∞	∞	1
4	∞	1	1	∞	1	1	∞	1	1
5	2	1	1	2	1	1	2	1	1
6	2	1	1	2	1	1	2	1	1
7	2	1	1	2	1	1	2	1	1

4 Fonction qui vérifie qu'une grammaire est correcte

Pour vérifier qu'une grammaire est correcte, on parcourt toutes les règles de la grammaire et pour chaque règle, s'il s'agit d'une règle de type ConstructorRule, on vérifie si les deux règles qu'elle possède en attributs sont définies dans la grammaire.

```
def check_defined_rules (grammar):
   for ruleId in grammar:
     rule = grammar[ruleId]
     if isinstance(rule, R.ConstructorRule):
       fst, snd = rule.parameters
       if not (fst in grammar):
         raise IncorrectGrammar(fst+"␣rule␣not␣defined␣in␣grammar")
       if not (snd in grammar):
         raise IncorrectGrammar(snd+"␣rule␣not␣defined␣in␣grammar")
```

Extrait de code 1 – Fonction check_defined_rules

5 Structure du programme

Le projet est composé de 7 fichiers, et a été développé avec Python 3.5 :

— Rules.py : contient les classes AbstractRule, ConstructorRule, ConstantRule, UnionRule, ProductRule, SingletonRule, EpsilonRule

— Grammars.py : contient la déclaration des grammaires de la question 2.2, des fonctions qui calculent des suites qui correspondent au nombre d'objets créés par une grammaire (ce point est expliqué plus en détail dans la partie suivante) ainsi qu'un dictionnaire qui associe à un nom de grammaire une liste qui contient la grammaire, la première règle de la grammaire et une fonction qui calcule la cardinalité.

— CRules.py : contient les classes permettant d'implémenter les grammaires condensées

— GrammarsC.py : contient les déclarations de grammaires condensées.

— Tree.py : contient la classe BinaryTree du TP3

— Project.py : contient les fonctions permettant d'initialiser les grammaires et les fonctions de tests

— Demo.ipynb : contenu similaire à Project.py sous forme de notebook pour la démonstration

6 Tests de cohérence génériques

Pour toute règle r d'une grammaire, pour tout entier n positif ou nul, les propriétés suivantes doivent être vérifiées :

— La longueur de la list générée par $r.list(n)$ doit être égale a $r.count(n)$

— Si on applique la fonction $r.rank$ à chaque élement de la liste générée par $r.list(n)$, on doit obtenir une liste égale à $list(range(r.count(n))$

— Si on applique la fonction $r.unrank$ à chaque élément de la liste $range(r.count(n))$, on doit obtenir une liste égale à $r.list(n)$

— On ne peut pas générer de mots ayant une taille inférieure à la valeur de la valuation de la règle

— Pour toutes les fonctions, pour toutes les règles, un paramètre négatif provoque une exception.

— Aucune règle ne doit avoir pour valuation ∞

— Si r est une *EpsilonRule* alors $count(n) = 0$ *si* $n \neq 0$ et $count(n) = 1$ sinon

— Si r est une *SingletonRule* alors $count(n) = 0$ *si* $n \neq 1$ et $count(n) = 1$ sinon

— Un appel à $r.unrank(n, i)$ avec $i \geq r.count(n)$ doit provoquer une exception

— Un appel à $r.rank(obj)$ où r est une règle de type $ConstantRule$ provoque une exception si obj n'est pas égal à l'objet passé en argument au constructeur de la règle.

— S'il existe une suite U_n qui dénombre les objets de la grammaire, alors $\forall n \geq 0$, on doit avoir $U_n = r.count(n)$ (les fonctions qui calculent ces suites sont dans le fichier Grammar.py).

ThreeGram : `https://oeis.org/A128588`
TreeGram : `https://oeis.org/A000108`
ABPalindrome : `https://oeis.org/A163403`
EqualGram : `https://oeis.org/A126869`
DyckGram : `https://oeis.org/A126120`
ABGram : 2^n

7 Ajout des fonctions : valuation, count, random, list, unrank

Pour le calcul des valuations d'une grammaire, on parcourt toutes les règles en mettant à jour les valuations pour les règles de type $ConstructorRule$ en boucle (puisque les valuations des règles de type $ConstantRule$ sont constantes) jusqu'à atteindre un point fixe. Les règles de type $ConstructorRule$ possèdent deux méthodes : get_valuation qui retoune la valeur de l'attribut valuation, et update_valuation qui met à jour la valuation.

Pour les autres fonctions (count, random, list, unrank), on implémente les algorithmes de l'énoncé en ajoutant des vérifications pour respecter la spécification décrite dans la partie précédente.

8 Ajout de la fonction rank

8.1 Ajout de rank dans Rules.py

Pour ajouter la fonction rank aux grammaires on modifie les constructeurs des classes UnionRule et ProductRule. On aurait pu ajouter le nouveau paramètre au constructeur de la classe ConstructorRule, mais les fonctions ajoutées ayant des signatures différentes en fonction du cas (Union ou Product), on préfère ajouter le nouveau paramètre à chaque classe, pour plus de cohérence.

Pour la classe UnionRule, le nouvel attribut est nommé $\_derive\_from\_first$, et correspond à une fonction qui retourne vrai si l'argument qui lui est passé en paramètre à été dérivé par la première règle qui compose l'union et faux sinon.

Pour la classe ProductRule, le nouvel attribut est nommé $\_get\_pair$, et correspond à une fonction qui prend un mot de la grammaire et le sépare en une paire dans laquelle le premier (resp. second) élément est la partie du mot dérivée par la première (resp. seconde) règle.

Afin de maintenir la possibilité de déclarer des grammaires sans avoir à écrire les fonctions de séparation, la fonction est un attribut optionnel du constructeur. L'appel à la fonction *rank* d'une règle pour laquelle la fonction n'a pas été fournie provoque une exception de type $NotImplementedError$.

Pour effectuer le calcul de *rank*, il est nécessaire d'avoir une méthode qui permet de calculer la taille d'un objet. Dans notre implémentation, nous utilisons la fonction python $\_\_len\_\_$ pour obtenir la longueur d'un objet. Il est donc nécessaire que les objets de la grammaire possèdent une définition pour $\_\_len\_\_$.

L'algorithme utilisé pour le calcul de $r.rank(obj)$ est :

— Si r est de type *EpsilonRule* ou *SingletonRule* alors si *obj* est égal à l'objet passé au constructeur, on retourne 0 sinon on lance une exception $ValueError$.

— Si r est de type $UnionRule(A, B)$, alors si *obj* dérive de A on retourne $A.rank(obj)$ sinon on retorune $A.count(len(obj)) + B.rank(obj)$.

— Si r est de type $ProductRule(A, B)$, avec $obj = (obj_{left}, obj_{right})$ alors on calcule le nombre d'objets $o = (o_{left}, o_{right})$ tels que $len(o_{left}) < len(obj_{left})$, auquel on ajoute le rang de *obj* dans la classe des objets de taille $(left, right)$ avec la formule $A.rank(o_{left}) + B.count(len(o_{right})) + B.rank(o_{right})$.

8.2 Exemple de fonctions à ajouter aux grammaires pour obtenir rank

Les fonctions spécifiques aux règles des grammaires pour calculer le rang ont été ajoutées dans le fichier Grammars.py. On présente ici, trois règles pour illustrer comment écrire ces fonctions.

```
# Regles extraites de la grammaire des mots de Fibonacci

"Fib"    : R.UnionRule("Vide", "Cas1", lambda s : s == "")

"CasAu" : R.ProductRule("AtomA", "Fib", "".join,
lambda s : s = (s[0], s[1:]))

# Regle extraite le la grammaire des mots de Dyck

"Casuu" : R.ProductRule("Dyck", "Cas(u", "".join, sep_dyck)

#avec

def sep_dyck (s):
  i_lpar = 0
  count = 0
  for i in range(len(s)):
      if s[i] == "(" and count == 0:
          i_lpar = i
          count += 1
      elif s[i] == "(":
          count += 1
      else:
          count -= 1
  return (s[:i_lpar], s[i_lpar:])
```

Extrait de code 2 – Exemple

— Pour la règle Fib, il est simple de distinguer les cas : si le mot est Vide, alors il a forcément été généré par la règle vide (car Cas1 ne permet pas de générer le mot vide).

— Pour la règle CasAu, on remarque que la partie générée par la règle de gauche est toujours AtomA, donc on peut séparer le mot après le premier caractère.

— Pour la règle Casuu, on remarque en observant la grammaire complète que le mot généré par "Cas(u" commence à la dernière parenthèse ouvrante parmis les parenthèses situées au niveau le plus 'extérieur' du mot.

9 Caching

Nous avons utilisé deux méthodes de caching différentes. La première est compatible avec python2 et consisite à écrire des fonctions de memoïsation dont le nom peut etre utilisé comme décorateur avant la déclaration de la fonction à mémoiser et la seconde est la solution proposée dans la version python 3.5 @lru_cache.

Le tableau suivant présente les durées d'exécution obtenues (moyenne sur 4 exécutions) pour la grammaire des arbres, avec des objets de taille 9.

Fonction	Sans mémoisation	Mémoisation Python2	Mémoisation Python3
count	0.0689	3.9517e-05	2.7775e-05
rank	40.1041	8.2734	33.1834
unrank	136.8276	26.4209	111.1100
rank (avec count memoïsé)		0.1138	0.08194
unrank (avec count memoïsé)		0.0178	0.02971

On remarque que la fonction ayant le plus d'impact sur les performances est la fonction count, ce qui s'explique par les nombreux appels que font rank et unrank à cette fonction. Les résultats illustrent l'intérêt d'utiliser le caching puisqu'il permet de réduire jusqu'à 100 fois le temps d'éxecution pour la fonction rank de cet exemple.

10 Ajout des Grammaires Condensées

10.1 Description Classe

Les classes permettant de construire des grammaires condensées sont contenues dans le fichier *CRules.py*. On reprend la structure de classe proposée dans l'énoncé en implémentant les classes suivantes : ConstantCRule, dont héritent Epsilon et Singleton, ConstructorCrule dont héritent Union et Prod, ainsi qu'une classe NonTerm.

Toutes les classes implémentent une fonction *to_simple_rule* qui permet de traduire la règle condensée en une règle développée.

Pour les classes Epsilon et Singleton, la fonction *to_simple_rule* prend en argument le dictionnaire auquel on va ajouter la règle et renvoie une paire composée du dictionnaire augmenté de la règle traduite et du nom de la règle ajoutée. La chaine de caractère choisie comme clé de la règle dans le dictionnaire correspond à la longueur du dictionnaire avant l'ajout de la règle.

Pour les classes Prod et Union, la fonction *to_simple_rule* prend en argument le dictionnaire et optionellement un nom pour la règle. On traduit chaque sous règle et on les ajoute au dictionnaire, puis crée une nouvelle règle à partir des règles traduites avec comme clé le nom fourni en argument (ou la taille du dictionnaire).

Pour la classe NonTerm, la fonction *to_simple_rule* ne modifie pas le dictionnaire car si $NonTerm("A")$ apparaît dans une règle, alors il doit y avoir une règle dans la grammaire condensée dont la clé est "A".

La fonction dvp_gram du fichier CRules.py prend en argument une grammaire condensée et retourne la grammaraire développée équivalente.

Les grammaires définies dans Grammars.py ont été réécrites sous forme de grammaires condensées dans GrammarsC.py (pour une question de lisibilité, dans certains cas, les règles ne sont pas totalement condensées).

10.2 Exemple traduction grammaire TreeGram

```
treeGram = {
"Tree" : CR.Union(

          CR.Prod(CR.NonTerm("Tree"), CR.NonTerm("Tree"),
          lambda l : Tree.Node(l),
          lambda t : (t.left(), t.right())),

          CR.Singleton(Tree.Leaf),
          lambda t: not (t.is_leaf()))
}
""" Au depart g = {}
Puis CR.Prod(CR.NonTerm("Tree"), CR.NonTerm("Tree"),
lambda l : Tree.Node(l),
lambda t : (t.left(), t.right()))
est ajoutee au dictionnaire :

g = {
  "0" : ProductRule("Tree", "Tree", lambda l : Tree.Node(l),
  lambda t : (t.left(), t.right()))
}
CR.Singleton(Tree.Leaf),
est ajoutee au dictionnaire
g = {
  "0" : ProductRule("Tree", "Tree", lambda l : Tree.Node(l),
  lambda t : (t.left(), t.right()))
  "1" : SingletonRule(Tree.Leaf)
}
Enfin, CR.Prod est traduite et ajoutee au dictionnaire
g = {
  "0" : ProductRule("Tree", "Tree", lambda l : Tree.Node(l),
  lambda t : (t.left(), t.right())),
  "1" : SingletonRule(Tree.Leaf),
  "Tree" : UnionRule("0", "1", lambda t : not (t.is_leaf()))
}
"""
```

Extrait de code 3 – Grammaire des arbres

11 Constructeur Bound

La classe Bound a été ajouté dans le fichier Rules.py. Le constructeur Bound prend en argument deux entiers min et max (avec min < max) et une règle r .

— $Bound.count() = \sum_{i=min}^{max} r.count(i)$

— $Bound.list() = \sum_{i=min}^{max} r.list(i)$ (ici, $\sum$ correspond à la concaténation de listes)

— $Bound.unrank(n) = r.unrank(j, n - \sum_{i=min}^{j-1} count(i))$ avec j le plus petit entier tel que $\sum_{i=min}^{j} count(i) \geq n$

— $Bound.rank(obj) = \sum_{i=min}^{len(obj)} r.count(i) + r.rank(obj)\ avec\ len(obj) < max$

— $Bound.random() = Bound.unrank(random(0, Bound.count))$

12 Sequence

Le constructeur Sequence a été ajouté aux grammaires condensées dans le fichier CRules.py. Il prend en argument un non terminal (chaine de caractères), un cas vide (objet), un constructeur (fonction), et optionellement deux fonctions pour rank (une pour l'union et une pour le produit). La fonction to_simple_rule effectue la traduction de la règle telle que décrite dans l'énoncé.

On réécrit la grammaire DyckGram en utilisant le nouveau constructeur :

```
DyckGram = {
  "Dyck" : CR.Union(CR.Epsilon(""), CR.Prod(CR.NonTerm("Dyck"),
  CR.NonTerm("Cas(u"), "".join)),
  "Cas(u" : CR.Prod(CR.Singleton("("), CR.NonTerm("Casu)"),
  "".join),
  "Casu)" :CR.Prod(CR.NonTerm("Dyck"), CR.Singleton(")"),
  "".join)
}
# devient
DyckGram = {
  "Dyck" : CR.Sequence ("Cas(u", "", "".join ),
  "Cas(u" : CR.Prod(CR.Singleton("("), CR.NonTerm("Casu)"),
  "".join),
  "Casu)" :CR.Prod(CR.NonTerm("Dyck"), CR.Singleton(")"),
  "".join)
}
```

Extrait de code 4 – Grammaire des mots bien parenthésés