# Analyse syntaxique par CKY probabiliste

# Table des matières

# I. Qu'est-ce que l'algorithme CYK ?

## CYK classique

Notre projet consiste en l'implémentation de l'algorithme CYK probabilisé.
L'algorithme CYK tire son nom des trois individus qui l'ont inventé, Cocke, Younger et Kasami. Il s'agit d'un algorithme tabulaire d'analyse syntaxique ascendante pour les grammaires non-contextuelles. Celui-ci permet de déterminer si un mot est reconnu ou non par une grammaire, si oui, il fournit également un arbre syntaxique.
Cet algorithme demande à ce que la grammaire soit en forme normale de Chomsky.
L'algorithme est de complexité O ( |g| * $n^3$ ), où |g| est la taille de la grammaire et n la longueur de la séquence à analyser.

Voici un exemple. Étant donné une grammaire en forme normale de Chomsky :

S -> SN SV
SN -> D N
SV -> V SN
V -> mange
D -> la | une
N -> fille | pomme

{S, SN, SV, V, N, D} est l'ensemble des non-terminaux et {mange, la, une, fille, pomme} est l'ensemble des terminaux, c'est-à-dire nos « lettres ».
Nous disposons également d'une séquence à parser : « la fille mange une pomme ».
Avec l'algorithme CYK, on obtiendra :

| | | | | |
|---|---|---|---|---|
| S | | | | |
| | | SV | | |
| SN | | | | |
| D | N | V | D | N |
| La | Fille | Mange | Une | Pomme |

## CYK probabiliste

Notre objectif était d'implémenter la version probabiliste de l'algorithme présentée précédemment.
Cette version probabiliste est intéressante car elle permet de choisir le ou les meilleures analyses syntaxique (et arbres syntaxiques). En effet, lorsqu'on remplit la demi-matrice avec une grammaire un peu plus complexe que le petit exemple vu précédemment, on a souvent des cases que l'on peut remplir avec plusieurs règles différentes. Or chaque règles peut finir par engendrer des analyses et des arbres parfois très différents les uns des autres. Avec la version probabiliste du parseur, on peut choisir de ne garder que la (ou les) analyse les plus probables, en choisissant de remplir les cases de la demi-matrice avec la (ou les) règles qui ont une meilleure probabilité.

Nous verrons dans la suite de ce documents quels choix d'implémentation ont été retenus mais également ceux qui ont été abandonnés et pourquoi.

# II. Réalisation du projet

Maintenant que nous avons vu ce qu'est l'algorithme CYK et sa version probabiliste, nous allons nous pencher sur notre implémentation de cet algorithme. Comme nous l'avons vu précédemment, CYK nécessite pour son fonctionnement une grammaire en forme quadratique (ou forme normale de Chomsky). Ici, comme nous souhaitons implémenter la version probabiliste, il faut en plus que chaque règle de notre grammaire ait une probabilité, c'est-à-dire qu'il nous faut une PCFG (Probabilistic Context-Free Grammar).

## Obtenir une grammaire pour le CYK

### Le corpus SEQUOIA

Pour obtenir cette grammaire, nous disposons d'un corpus contenant plus de 3 000 phrases en français. Il s'agit du corpus Sequoia composé de diverses sources (Europarl, Le Monde, Wikipédia et EMA), c'est donc un corpus aux thèmes assez variés.
Un corpus à thèmes variés est meilleur qu'un corpus sur un seul thème ou domaine. Par exemple si l'on extrait une grammaire uniquement à partir d'un corpus médical, il y aura beaucoup de productions de la langue française qui nous échapperont totalement. En effet dans ce genre de corpus scientifique, il n'y a pas de dialogues, il n'y a pas ou peu de phrases à la première ou à la seconde personne et le vocabulaire est assez particulier et restreint. Des corpus très varié permettent lors de l'apprentissage de couvrir un maximum de phénomènes de la langue dans toute sa variété.
Le corpus Séquoia nous fournit un grand ensemble de phrases au format MRG. Ce format parenthésé est très pratique pour être parsé.
Voici un exemple de phrase que l'on peut trouver dans le corpus :

((SENT (NP-SUJ (DET La) (NC tâche)) (VN (ADV ne) (V sera)) (ADV pas) (AP-ATS (ADJ aisée)) (PONCT .)))

Pour extraire une grammaire à partir de ce corpus, on doit d'abord définir quels seront nos terminaux et nos non-terminaux. Nos terminaux seront les mots du lexique, et nos non-terminaux sont les étiquettes morphosyntaxiques.
Nous avons décidé d'ignorer les fonctions grammaticales car elles ne sont pas primordiales pour un parseur syntaxique mais surtout parce qu'elles viendraient augmenter la taille de la grammaire et donc augmenter la complexité en temps de l'algorithme (rappelez-vous, nous avons vu dans la présentation du CYK que la complexité en temps de ce dernier est proportionnelle à la taille de sa grammaire).

## Programme d’extraction

Notre programme d’extraction de grammaire prend en entrée le fichier MRG.
L’objectif étant d’obtenir en sortie une grammaire hors contexte probabilisée en forme normale de Chomsky.

## Probabiliser la grammaire

Comment s’y prendre pour probabiliser nos règles de réécriture ? C’est une question sur laquelle nous avons passé du temps. Nous avions deux stratégies et nous avons dû faire un choix. Tout d’abord, nous avions pensé à probabiliser chaque règle au fur et à mesure qu’elles soient extraites. Cela semblait plus simple au départ. Notre idée était la suivante, chaque fois qu’on rencontre une règle :

- Si la partie gauche n’a jamais été trouvée avant, la règle vaut 1.
- Si la partie gauche a déjà été trouvé avant mais que la partie droite est nouvelle, on divise 1 par le nombre de règles vu auparavant (+1 car on vient d’en trouver une autre) et on obtient la probabilité de chaque règle partageant cette même partie gauche. Il faut alors mettre à jour toutes ces règles-là.
- Si la partie gauche a déjà été trouvée et que la partie droite aussi (c’est-à-dire qu’on a déjà rencontré cette même règle) alors on divise 1 par le nombre de règles (comme précédemment) sauf qu’ensuite on additionne les probabilités des deux règles identiques et on en supprime une. Ainsi la règle qui a été vu deux fois (ou plus) a une probabilité plus forte que les autres qui n’ont été vues qu’une seule fois.

Finalement, nous avons décidé de probabiliser chaque règles après qu’elles soient toutes extraites. En effet, cela évite de faire sans cesse des mises à jour des règles vues auparavant. La méthode est la suivante :

Pendant l’extraction des règles, on dispose de compteurs pour chaque parties gauche et chaque parties droite de règles, qui sont incrémentés chaque fois que ces éléments sont rencontrés dans le corpus.

À la fin du parcours,

soit n le nombre de fois qu'une partie gauche N a été comptabilisée, et m le nombre de fois qu'elle se récrit en une partie droite M, la probabilité d'une réécriture $N \rightarrow M$ est de m/n.

On assigne donc pour chaque réécriture M de chaque non-terminal N la probabilité m/n correspondante.

Pour mieux comprendre ce processus, voici un exemple avec une petite grammaire :

VP -> v
VP -> v NP
VP -> v ADJ
VP -> v

On dispose de 4 règles qui ont la même partie gauche « VP » donc notre compteur de VP est à 4. Nous disposons également de 2 règles VP identiques, donc notre compteur de règles « VP -> v » est à 2.
Nous attribuons une probabilité à chacune des règles en divisant 1 par le nombre total de règles, nous obtenons donc une probabilité de 1/4 pour chaque règle.

VP -> v = 1/4
VP -> v NP = 1/4
VP -> v ADJ = 1/4
VP -> v = 1/4

Puis on additionne les probabilités de toutes les règles en double et supprime l'une des deux règles identiques pour ne plus avoir de doublons :

VP -> v = 1/4 + 1/4 = 1/2
VP -> v NP = 1/4
VP -> v ADJ = 1/4

Dans un souci de précision, nous avons choisi d'utiliser le module « fractions » pour Python pour les probabilités, et non pas des float.

## Mettre la grammaire en forme normale de Chomsky

Une grammaire est en forme normale de Chomsky si l'axiome S est inaccessible, et si toutes les règles de production sont de la forme A => B C ou D => e ou S => Ɛ, avec A, B, C et D des non-terminaux quelconques, e un terminal quelconque et Ɛ la production vide.

Par conséquent, la transformation d'une grammaire en grammaire CNF faiblement équivalente comporte les étapes suivantes :

- Faire en sorte que l'axiome n'apparaisse plus dans les parties droites de règles.
- Supprimer les règles d'effacement (c'est à dire de la forme A => Ɛ) pour les non-terminaux autres que l'axiome.
- Faire en sorte que tous les terminaux apparaissent uniquement dans la partie droite de règles unaires.
- Remplacer les règles de production n-aire par des règles binaires équivalentes
- Supprimer les productions singulières de non-terminaux, c'est à dire les règles de la forme A => B avec A et B non-terminaux.

Étant donné que nous construisons la grammaire à partir d'un corpus déjà étiqueté et parsé, nous n'avons pas eu besoin d'implémenter les trois premières étapes : le corpus SEQUOIA est déjà construit de manière à ce qu'il n'y a pas de non-terminal sans descendant, donc pas de règle d'effacement, l'axiome SENT n'a jamais de parent, et tous les terminaux sont déjà exclusivement enfants uniques d'un non-terminal.

Par conséquent, seules les deux dernières étapes de cette transformation ont été effectivement implémentées.

Une contrainte supplémentaire a pesé sur notre implémentation de la conversion en CNF : la nécessité de pouvoir transformer les arbres obtenus avec notre parser en arbres correspondant à la grammaire originale.

Notre référence principale (Roark & Sproat) propose une implémentation où l'étape de suppression des productions singulières est effectuée avant la binarisation. Or, si l'étape de binarisation provoque un accroissement linéaire de la taille de la grammaire, l'étape de désingularisation provoque quant à elle un accroissement exponentiel, de l'ordre de $2^n$ règles supplémentaires où n est dans le pire des cas la longueur d'une partie droite de règle.
On a donc tout intérêt à binariser avant de supprimer les productions singulières, ainsi que le fait remarquer l'article de Leiß et Lange. D'autant plus que la complexité algorithmique de CYK dépend linéairement de la taille |G| de la grammaire qu'il utilise.

Pour binariser :

Pour toutes les règles A => B γ où γ est une suite quelconque de non-terminaux, on remplace cette règle par une règle A => B Γ qui prend la probabilité de A => B γ où Γ est un non-terminal qui représente la suite de non-terminaux γ, et on introduit une nouvelle règle Γ => γ de probabilité 1.

On répète cette étape jusqu'au point fixe (c'est à dire jusqu'à ce que cette étape ne change plus rien).

Pour supprimer les productions singulières, on répète toutes ces étapes jusqu'au point fixe :

- Pour toutes les règles de la forme A => B avec une probabilité p1, on introduit un nouveau terminal AB.
- Pour toutes les règles de la forme B => μ (où μ est une suite quelconque de non-terminaux), on introduit une règle de la forme AB => μ, avec la même probabilité.
- Puis, pour toutes les règles de la forme C => αAβ ayant une probabilité p2, on introduit une règle de la forme C => αABβ avec une probabilité p1*p2 et on change la probabilité de C => αAβ en p2*(1-p1).

C’est cette dernière étape qui peut provoquer une explosion de la taille de la grammaire puisque si les règles sont encore n-aires alors la partie droite peut contenir un nombre arbitraires de A, qui peuvent tous être remplacés par des AB indépendamment les uns des autres.)

## Notre implémentation du CYK

Notre implémentation du CYK va suivre le principe général du CYK exposé plus haut, avec cette différence qu'en pratique, pour parcourir la grammaire et remplir les cases, on préfère utiliser une table de hachage (dictionnaire python) adressée par les parties gauches puis droites des règles, de sorte qu'on trouvera plus rapidement les non-terminaux susceptibles de récrire les cases, que si on parcourait intégralement la grammaire à chaque étape.

D'autre part, le fait que chaque règle soit associée à une probabilité permet de récupérer la probabilité maximale de chaque réécriture et donc de diminuer le nombre d'éléments présents dans chaque case.

Une fois la demi-matrice du CYK remplie jusqu'à la dernière case, il n'y a ensuite plus qu'à faire du backtracking, c'est à dire récupérer parmi les versions de l'axiome présent dans la dernière case, celle qui a la meilleure probabilité et qui contient des pointeurs vers les cases correspondants aux enfants de l'axiome, et de descendre ainsi récursivement dans la demi-matrice en récupérant les constituants ayant les meilleures probabilités, d'une manière similaire à l'algorithme de Viterbi.

Malheureusement, pour des raisons de performance, notre CYK convertit en flottants les fractions exactes utilisées par les étapes d'extraction et de transformation de la grammaire.

Cela peut susciter un échec du parsing pour les phrases très longues (plus de 120 mots) et très ambiguës.

### Remettre la grammaire en forme n-aire

Afin d'évaluer notre parseur, il nous faut retransformer les arbres binaires qu'il produit en arbre correspondant à la grammaire plate utilisée dans le corpus. Pour cela on utilise le fait que les non-terminaux introduits à l'étape de transformation en CNF sont marqués comme tels, et contiennent les informations nécessaires à la reconstitution des arbres correspondant. On va donc reconstituer les productions singulières à partir des non-terminaux comportant l'indication qu'ils correspondaient à des productions singulières en descendant récursivement dans l'arbre, puis parcourir l'arbre obtenu en concaténant les enfants d'un non-terminal Γ correspondant à une réécriture n-aire au niveau de son nœud frère.

## III. L'évaluation

### Généralités

- **Rappel** : Le rappel est le ratio du nombre de constituants correctement prédits par rapport au nombre de constituants gold.

- **Précision :** La précision est le ratio du nombre de constituants correctement prédits par rapport au nombre total de constituants prédits.

- **F-score :** Le f-score, aussi appelé f-mesure, est la moyenne harmonique de la précision et du rappel.

$$\text{F-score} = \frac{2 \times (\text{Précision} \times \text{Rappel})}{(\text{Précision} + \text{Rappel})}$$

Dans l'évaluation on considère qu'un constituant est correctement prédit s'il à la même position de début et de fin dans la phrase qu'un constituant gold, ainsi que la même étiquette dans le cas de l'évaluation étiquetée.

Cela correspond à la métrique de PARSEVAL, décrite en premier par Black, Abney et Al. (voir Annexes).

## Notre programme d’évaluation

Le script prend en input des arbres prédits par le CYK et des arbres gold.
Une fois assuré que la phrase prédite est la même que la phrase gold, nous comptabilisons le nombre de constituants communs aux deux phrases et les constituants spécifiques à la gold et à la prédiction que nous conservons dans des variables utilisées par la suite pour le calcul de la précision, du rappel et de la f-mesure de chaque phrase.
Pour chaque phrase, le script soustrait au nombre de constituants prédits correctement le nombre de feuilles, car ces dernières sont forcément prédites correctement.
En revanche, il comptabilise tout de même le constituant maximal SENT, y compris lorsqu’il s’agit de l’unique constituant prédit (ce qui arrive en cas d’échec du parseur).
Le script calcule d’une part la moyenne des précisions, rappel et f-mesure des différentes phrases et d’autre part, il calcule la précision, le rappel et la f-mesure sur l’ensemble des constituants prédits par le CYK. Le script peut faire ce calcul pour des constituants étiquetés ou non (voir Manuel d’utilisation).

## Les résultats obtenus

Voici les résultats qui ont été obtenus sur le corpus de test avec la grammaire créée à partir du corpus d’entraînement situé dans le dossier test. Ces mesures ont été calculées en tenant compte des étiquettes.

Rappel global : 0.8277998705262184
Précision globale : 0.8294106745737584
F-mesure globale : 0.8286044897014581

# IV. Manuel d'utilisation

## Prérequis

Tous les scripts ont été écrits en Python 3, de ce fait, nous ne garantissons pas qu'ils fonctionnent en Python 2.
Nous avons également utilisés quelques modules python, qui font a priori partie de la librairie standard de python.
En voici la liste complète :

- Collections
- Copy
- Optparse
- Pickle
- Argparse
- Codecs
- Fileinput
- Fractions
- Logging
- Pickle
- Random
- Re
- Sys
- Codecs

NB : Les scripts ont des dépendances les uns dans les autres, il vaut donc mieux les garder dans le même répertoire.

## Quelques exemples de commandes

Cette section présente de façon rapide l'ensemble des scripts du projet CYK probabiliste.
En lançant les commandes présentées dans l'ordre, on obtient successivement un corpus d'entraînement et de test, une grammaire tirée du corpus d'entraînement un ensemble de parse candidats et enfin leur évaluation vis-à-vis du corpus de test.

Étant donné un fichier mrg fi.mrg :
La commande :

```
dispatch.py fi.mrg train.mrg test.mrg
```

crée deux fichiers,

- train.mrg, qui doit servir de corpus d'entraînement (90 % de fi.mrg)
- test.mrg qui doit servir de corpus de test (10% de fi.mrg )

(La répartition entre les deux fichiers est faite au hasard).

La commande

```
extracteur.py fi.mrg grammaire.pickle
```

extrait une PCFG (sous forme normale de Chomsky) à partir de fi.mrg et l'enregistre dans le fichier grammaire.pickle.

```
extracteur.py train.mrg grammaire_train.pickle
```

extrait une PCFG à partir de train.mrg et l'enregistre dans le fichier grammaire_train.pickle.

```
updategrammar.py grammaire.pickle grammaire_train.pickle
```

modifie le fichier grammaire_train.pickle : elle sert à transférer dans la grammaire d'entraînement l'ensemble des règles lexicales du corpus afin d'éviter d'avoir des erreurs dues à des mots inconnus.

La commande

```
ckys.py grammaire_train.pickle test.mrg > sortie_test.mrg
```

lance l'analyse des phrases de test.mrg par CYK probabiliste et les rassemble dans sortie_test.mrg.

ATTENTION : cette étape peut durer extrêmement longtemps (plusieurs heures). Pour voir rapidement le fonctionnement de cky, utiliser le "mode interactif" (cf. ci-dessous).

La commande

evaluation.py --gold test.mrg sortie_test.mrg

calcule précision, rappel et f-mesure non-étiquetés sur sortie_test.mrg en prenant pour référence les parses originaux dans test.mrg.

## Les commandes du CYK

### Commandes principales

Le script ckys.py se lance avec deux arguments.
Le premier est un pickle contenant une PCFG sous forme normale de Chomsky produite par le script extracteur.py.
Le deuxième argument est un corpus, un fichier mrg contenant les phrases que le script devra analyser.

ckys.py grammaire_train.pickle test.mrg

Par défaut, le script est lancé de façon non-interactive : il traite toutes les phrases du corpus en commençant par la première et imprime l'analyse sur stdin et des messages d'informations ou d'erreur sur stderr.

### Les options

- L'option –i

  L'option -i permet de lancer le script en mode interactif. Le mode interactif affiche chaque phrase une par une avec leur longueur et leur numéro, at attend des commandes de l'utilisateur. Les commandes acceptées par le script sont les suivantes:

  - exit
  - quit
  - Ctrl+D : quitte le script.

  - y : lance l'analyse de la phrase courante et l'affiche. Durant l'analyse, il est possible de l'arrêter en utilisant le raccourci Ctrl+C

  - goto NOMBRE : va à la phrase numéro NOMBRE si NOMBRE est plus grand que le numéro de la phrase actuelle.

- L'option -p

  L'option -p donne la position de départ dans le corpus du script. Par défaut le script commence au début du fichier. Spécifier un numéro avec l'option -p permet de le faire commencer à la phrase correspondant à ce numéro.

## Le script extrateur.py

### Commandes principales

Extracteur.py est un script qui permet d'extraire une grammaire probabilisée sous forme normale de Chomsky, à partir d'un fichier au format mrg (mrg strict ou id mrg).
Le script extracteur.py se lance avec deux arguments :

- Le nom du fichier mrg à partir duquel construire la grammaire.
- Le nom du fichier où sauvegarder cette grammaire.

La grammaire ainsi crée est un fichier pickle (objet python sérialisé) contenant un tuple de trois éléments :

- L'ensemble (set) des terminaux de la grammaire.
- L'ensemble des non-terminaux
- Un dictionnaire contenant les règles de productions et leur probabilité.

## Le script evaluation.py

### Commandes principales

Evaluation.py est le script qui calcule la précision, le rappel et la f-mesure étiquetés ou non pour les constituants d'un ensemble d'analyses lues depuis un fichier, depuis un stdin, ce qui permet de l'utiliser avec un pipe unix.
Les analyses gold doivent être obligatoirement lue depuis un fichier spécifié avec l'option –g ou –gold.
Pour l'utiliser avec  un fichier, il faut le lui passer en argument :

Evaluation.py –g fichierGold.mrg candidats.mrg
Ou
Evaluation.py candidats.mrg –g fichierGold.mrg
Ou
Cat candidats.mrg | evaluation.py –g fichierGold.mrg
Ou même
Ckys.py grammaire.pickle fichierGold.mrg | evaluation.py –g fichierGold.mrg

### Les options

- L'option –l

  Si l'option –l ou –labeled est activée, les constituants seront considérés comme correct uniquement s'ils ont e même span ET la même étiquette. Sinon, seul le span sera considéré.

- L'option –v

  L'option –v ou –verbose affiche la précision, rappel et f-mesure pour chaque analyse, ainsi que les span manquant du gold ou les span suplémentaire.

# V. Annexes

## Listing des algorithmes

**Algorithm 1** CNF

**function** CHOMSKYNORMALFORM($G :< NT, T, \rho >$) ▷
$G$ : Grammaire, $\rho$ : productions probabilis
  $P' = \rho.copy()$; ▷ $\rho$ est un dictionnaire de dictionnaire de fractions
  **function** BINARISER($nt, p, proba$)
    **if** |p| $> 2$ **then**
      $nt^{\alpha} \leftarrow join(\downarrow, p[1:])$;
      $NT = NT | nt^{\alpha}$;
      $P'[nt][(p[0], nt^{\alpha})] \leftarrow proba$;
      BINARISER($nt^{\alpha}, p[1:]$);
    **else**
      $P'[nt][p] \leftarrow proba$;
  **for** $nt \in \rho$ **do**
    **for** $p \in \rho[nt]$ **do**
      **if** $|p| > 2$ **then**
        BINARISER($nt, p, \rho[nt][p]$);
        $cnf = cnf - cnf[nt][p]$;
  **for all** $nt^A \rightarrow nt^B; \rho^1 \in P$ **do**
    $nt^{\alpha}$ = JOIN($\uparrow, nt^A, nt^B$);
    $NT \uplus nt^{\alpha}$;
    **for all** $nt^C \rightarrow \alpha, nt^A, \gamma; \rho^2 \in P$ **do**
      $cnf \uplus cnf[nt^C][(\alpha, nt^{\alpha}, \gamma)] = \rho^1 * \rho^2$;
      $cnf[nt^C][(\alpha, nt^A, \gamma)] = (1 - \rho 1) * \rho^2$;
    **for all** $nt^A \rightarrow \alpha; \rho^3 \in P$ **do**
      $\rho^3 = \rho^3 / (1 - \rho^1)$;
    **for all** $nt^B \rightarrow \alpha; \rho^4 \in P$ **do**
      $cnf \uplus cnf[nt^{\alpha}][(\alpha)] = \rho^4$;
  **return** $G' :< NT, T, P' >$

**Algorithm 1** CKY probabiliste (Max Product)

**function** CKY($w[1..n]$, $G :< NT, T, P, \rho >$, R[1..n, 1..n]) ▷ w : mot ; G : Grammaire ; C : charte
  **for all** $max \leftarrow 2, |n|$ **do** ▷ Boucle gnt l'empan
    **for all** $min \leftarrow max - 2, 0$ **do**
      **for all** $nt \in NT$ **do**
        $best = 0$;
        **for all** $nt \rightarrow nt^1 nt^2 \subset P$ **do**
          **for all** $mid \leftarrow min + 1, max - 1$ **do**
            $t1 = R[min][mid][nt^1]$;
            $t2 = R[mid][max][nt^2]$;
            $candidate = t1 * t2 * \rho(binary)$;
            **if** $candidate > best$ **then**
              $best = candidate$;
        $R[min][max][nt] = best$;

**Algorithm 1** Débinariation

```
function BINAIRE2NAIRE(tree : list)
    tmp ← LIST(;)
    bin ←↓;
    for feuille ∈ tree do
        if ISINSTANCE(feuille, list) then
            if bin ∈ feuille[0] then
                tmp.extend(UNBIN(feuille⌊1 :⌋));
            else
                tmp.append(UNBIN(feuille))
        else
            tmp.append(feuille)
    return tmp
function LEXICALE2UNAIRE(tree)
    if |tree| == 3 then
        head, leftchild, rightchild ← tree;
        if ↑∈ head then
            gauche, droite ← SPLIT(bin, head, 1)
            return [gauche, GENERALUNBIN(⌊droite, leftchild, rightchild⌋)]
        else
            return [head, LEXICALE2UNAIRE(leftchild), LEXICALE2UNAIRE(rightchild)]
    else if |tree| == 2 then
        head, child ← tree;
        if ↑∈ head then
            gauche, droite ← SPLIT(bin, head, 1)
            return [gauche, LEXICALE2UNAIRE(⌊droite, child⌋)]
        else
            return [head,child]
function GENERALUNBIN(tree)
    firststep ← LEXICALE2UNAIRE(tree)
    return GENERALUNBIN(firststep)
```

## Bibliographie

Martin LANGE & Hans LEIß, *To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm*, 2009, Zeitschrift für fachdidaktische Grundlagen der Informatik, https://www.informaticadidactica.de/index.php?page=LangeLeiss2009
Brian E. ROARK & Richard SPROAT, *Computational Approaches to Morphology and Syntax*, Oxford University Press, 2007.
Mariana ROMANYSHYN & Vsevolod DYOMKIN, *The Dirty Little Secret of Constituency Parser Evaluation*, 2014, http://tech.grammarly.com/blog/posts/The-Dirty-Little-Secret-of-Constituency-Parser-Evaluation.html.
Ivan TITOV, 2000, http://ivan-titov.org/teaching/nlmi/
Page Wikipédia, *Précision et rappel*, https://fr.wikipedia.org/wiki/Pr%C3%A9cision_et_rappel

Alain BACCINI, Sébastien DEJEAN, Désiré KOMPAORE, Josiane MOTHE, *Analyse des critères d'évaluation des systèmes de recherche d'information*, https://www.irit.fr/publis/SIG/2010_ISI_BADSKNDMJ.pdf

E. Black, S.Abney et al. , *A Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars*, 1991, DARPA Speech and Natural Language Workshop, pp.306-311. Morgan Kaufmann