université PARIS-SACLAY

FACULTÉ DES SCIENCES D'ORSAY

Cours de PIL, grammaires 2022-2023 Frédéric Gruau

1 Grammaire.

- Décrit des langages
- Inspirée des grammaires de langage naturel, exemple : Phrase → sujet verbe complément.
- Différence : une grammaire peut générer des mots arbitrairement long, les phrases restent de longueur raisonnable.

On commence par donner une définition générale des grammaires :

Définition 1 *Une grammaire G est un quadruplet (Σ_N,Σ_T, S, R) où*

1. Σ_T *= alphabet de symboles dits terminaux ;*
2. Σ_N *alphabet de symboles non-terminaux ;*
3. $\Sigma = \Sigma_N \cup \Sigma_T$ *= alphabet total de G ;*
4. *S est un non-terminal appelé axiome ;*
5. $R \subseteq \mathcal{P}(\Sigma_N \times \Sigma^\star)$ *est un ensemble fini de règles notées* $g \to d$ *si* $(g, d) \in R$.

Notation : On utilisera des majuscules pour les non-terminaux, et des minuscules pour les terminaux.

Exemple1 La grammaire $S \to aSBC, S \to \epsilon, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc$.

Faire une dérivation, montrer que cela génère $a^n b^n c^n, n > 0$. Notation : on utilise un trait verticale pour regrouper les différents membres droit associés à un même membre gauche. Dans la grammaire ci-dessus, on peut ainsi noter $S \to aSBC|\epsilon$

1.1 Langage engendré.

On engendre les mots du langage, en récrivant un mot $u \in \Sigma^\star$ en un nouveau mot $v \in \Sigma^\star$. On remplace une occurrence (quelconque) d'un membre gauche de règle présent dans u par le membre droit de cette règle.

Définition 2 *Étant donnée une grammaire* $G = (\Sigma_T , \Sigma_N , S, R)$*, on dit que le mot* $u \in \Sigma^\star$ *se récrit en le mot* $v \in \Sigma^\star$ *dans G avec la règle* $g \to d$*, et on note* $u - g, d \to v$ *si* $u = w_1 g w_2$ *,* *et* $v = w_1 d w_2$ *;*

On peut aussi noter plus simplement $u \to v$. Plus généralement, on dit que le mot $v \in \Sigma^\star$ dérive du mot $u \in \Sigma^\star$, dans la grammaire G, et on note $u \to^* v$, (fermeture transitive de $\to$) s'il existe une suite finie $w_0, w_1, \ldots, w_n$ de mots de $\Sigma^\star$ telle que $w_0 = u, w_i \to w_{i+1}$ pour tout $i \in 0, \ldots N-1$, et $w_n = v$. On peut indiquer le non-terminal récrit en le soulignant. La réécriture est itérée à partir de l'axiome jusqu'à l'élimination complète des non-terminaux.

Définition 3 *Le langage engendré par la grammaire G est l'ensemble des mots de* $\Sigma_T^\star$ *qui dérivent de l'axiome de G, que l'on note par Lang(G)ou L(G).*

Classification de Chomsky :

- type 0 : membre gauche arbitraire membre droit arbitraire (Machine de Turing)
- type 1 : on passe
- type 2, hors contexte : membre gauche = un seul non terminal (Automate à Pile)
- Type 3, régulier : membre droit contient un seul non terminal toujours tout à la fin (Automate d'état fini)

Exemple2 (grammaire régulière) : La grammaire $N \to 0|1M, M \to 0M|1M|\epsilon$ génère les entiers naturels en représentation binaire, sans zéros redondants : par exemple, 01 n'est pas dedans.

1.2 Grammaires hors-contexte.

Les grammaires régulières type 3 sont trop simple ; les langages que elles génèrent sont les langages rationnels, pourquoi s'ennuyer à définir un outil puissant pour rester dans ce monde limité ?

Les grammaires type 1 sont trop compliquées pour nous, elles peuvent générer n'importe quel

langages dés l'instant ou il existe un algorithme qui peut le faire.

On va se concentrer exclusivement sur les grammaires hors contexte, de type 2, elle sont suffisement puissantes pour décrire la syntaxe des langages de programmations, et suffisement simple pour autoriser automatiquement une analyse de cette syntaxe. On dit "hors-contexte", car le non-terminal du membre gauche décide tout seul (sans contexte) comment il souhaite se réecrire.

La grammaire de l'exemple 1 n'est pas hors-contexte, car les membres gauches contiennent plusieurs lettres. Exemple3 (Grammaires hors-contexte) La grammaire $S \to \epsilon|aSb$ génére le langage $\{a^n b^n | n \geq 0\}$.

Si la grammaire est hors-contexte, le langage généré est dit ALGEBRIQUE. On considère seulement ceux-la dans la suite. Ils incluent tout les langages de programmation usuels.

1.3 Arbre de dérivation

La propriété hors contexte permet de représenter une dérivation par un arbre. C'est bien le levier qui va nous permettre de construire une théorie simple.

Considérons la grammaire hors-contexte suivante : $\Sigma_T = \{+, *, (,), Id, Cte\}, \Sigma_N = \{E\}, E \to Id|Cte|E + E|E * E|(E)$. En vrai, Cte et Id représentent des constantes ou des identificateurs. Pour l'analyse syntaxique, on considère toutes les constantes (resp. tout les identificateurs) comme le même terminal. Faire deux arbres de dérivation distincts de $x + 4 * y$, montrer que le sens diffère ;

Définition 4 *Étant donnée une grammaire G = (Σ_T , Σ_N , S, R), les arbres de dérivation de G sont des arbres avec la racine (resp. les nœuds internes, les feuilles) étiqueté(es) par l'axiome, (resp. des non terminaux, des terminaux) vérifiant de plus que si les fils pris de gauche à droite d'un nœud interne étiqueté par le non- terminal N sont étiquetés par les symboles respectifs $\alpha_1, ..., \alpha_n$, alors $N \to \alpha_1 \dots \alpha_n$ est une règle de la grammaire G.*

Un arbre de dérivation résume plusieurs dérivations possibles, réalisées avec un ordonnancement différent.

1.4 Ambiguïté

Deux arbres différents, cela implique un non-déterminisme, et aussi deux calculs différent. On n'aime pas, on va définir l'ambiguïté comme suit, et chercher ensuite à l'éviter.

Définition 5 *Une grammaire G est ambiguë s'il existe un mot $w \in Lang(G)$ qui possède plusieurs arbres de dérivation dans G.*

Démontrer l'ambiguïté est facile, il suffit d'exhiber deux arbres de dérivation pour un mot. Par contre, démontrer qu'une grammaire n'est pas ambiguë est difficile et considéré comme hors programme. Cela requiert une démonstration par récurrence sur la profondeur des arbres de dérivations. En TD, on se convaincra de la non-ambigüité d'une grammaire en construisant l'arbre de dérivation d'un mot représentatif, et en constatant qu'on n'a jamais de choix durant cette construction.

Comment résoudre l'ambiguité ?(TD)
Méthode 1, bricolage : on introduit d'autre non-terminaux pour forcer un ordre : $E \to E + T|T$
$T \to T * F|F$
$F \to id|cte|(E)$
Méthode 2, la méthode utilisée en pratique, car plus simple, et plus élégante. On utilise des méta règles, extérieures à la grammaire :

- priorité de * sur +
- associativité à gauche de $*, +$.

Définition 6 *Réécriture droite (resp gauche) on réécrit le non-terminal le plus à droite (resp. à gauche)*

La réécriture droite (resp. gauche) corresponds a un parcours droit (resp. gauche) de l'arbre de dérivation. On parle aussi de dérivation droite et dérivation gauche. Il y a une correspondance bi-univoque entre dérivation gauche (resp. droite) et arbre de dérivation.

Propriété : une grammaire G est non ambigüe si et seulement si tout mot a une seule dérivation droite (resp. gauche).

1.5 Clôture algébriques

Les langages algébriques sont clos par

- clos par union $S \to S_1|S_2$
- clos par étoile $S \to SS_1|\epsilon$
- Pas clos par intersection !.

— Pas clos par complémentaire ! !.
— Clos par intersection avec rationnel

Attention, l'intersection de deux langages n'est pas algébrique contre exemple $a^n b^n c^m$ et $a^m b^n c^n$ intersecté donne $a^n b^n c^n$. Le complémentaire d'un algébrique, n'est pas algébrique, sinon l'intersection le serait. En effet on peut exprimer l'intersection à partir de l'union et du complémentaire : $A \cap B =$ complémentaire (complémentaire $(A) \cup$ complémentaire (B))

2 Analyse CYK

On s'intéresse au problème de l'analyse syntaxique qui consiste étant donné un mot m est une grammaire G à trouver si m peut être engendré par G et du même coup à donner un arbre de dérivation de m. Notre premier algorithme d'analyse syntaxique marche seulement sur les grammaires propres, et plus précisément sous forme FNC. On commence par expliquer ces deux trucs. L'une des raisons de présenter CYK est que c'est un exemple de programmation dynamique, et cela recouvre un grand nombre d'applications. On illustre l'algorithme sur trois petits exemples simple de grammaires.

2.1 Nettoyage de grammaire.

Définition 7 *Une grammaire hors-contexte G = (Σ_T , Σ_N , S, R) est dite propre si elle vérifie :*

1. $\forall N \to u \in R, u \neq \epsilon$ *ou* $N = S$
2. $\forall N \to u \in R$, *On n'a pas de S dans u*
3. *Les non-terminaux sont tous utiles, c'est-à-dire à la fois atteignables et productifs.*
4. *Il n'y a pas de règle oú on remplace un non terminal par un autre.*

Un non-terminal est dit atteignable si on peut le générer depuis l'axiome, il est dit productif s'il peut générer une chaîne de terminaux. Donner des exemples négatifs pour illustrer ;

Theorème 1 *On peut toujours rendre une grammaire propre*

On passe sur la preuve cela prend du temps, ce sera vu l'an prochain.

2.2 Forme normale de Chomsky

On va l'appeler "FNC" dans la suite. Cette forme de grammaire est necessaire pour faire fonctionner "CYK".

Theorème 2 *Pour tout langage algèbrique L, il existe une grammaire propre G qui l'engendre dont toutes les règles sont de l'une des trois formes $S \to \epsilon, P \to a,$ ou $P \to MN$, avec M, N différents de S. (c'est la FNC Forme Normale Chomsky)*

Preuve : Cette preuve là est facile. on la fait. Partant d'une grammaire hors-contexte propre $G = (\Sigma_T , \Sigma_N, S, R)$ on aplique deux étapes :

1. On rajoute une copie de Σ_T à Σ_N (on notera A la copie de a), puis l'ensemble de règles $A \to a | a \in \Sigma$.
2. On ne touche pas aux règles dont le membre droit est juste un terminal.
3. Pour les autres, on remplace les symboles terminaux a figurant dans les membres droits de règles initiales par le non-terminal correspondant A, puis on remplace la règle $X \to X_1 ... X_n$ pour $n > 2$ par $X \to X_1 Y$ et $Y \to X_2 ... X_n$ en ajoutant un nouveau non-terminal Y. On obtient ainsi une nouvelle règle, avec un membre droit ayant un non-terminal en moins. On itère cette opération qui rajoute à chaque fois une nouvelle règle, mais avec de moins en moins de non terminaux en membre droit, jusqu'à temps qu'il n'y ait plus que deux non-terminaux en membre droit.

Exemple : notre fameuse grammaire $S \to \epsilon, S \to aSb$ qui génère $\{a^n b^n, n \geq 0\}$. On choisit celle-là car elle est très simple et suffit pour illustrer. Il faut commencer par lui appliquer l'algorithme de nettoyage, car elle n'est pas propre : en effet l'axiome apparaît en partie droite. Celui-ci donne la grammaire propre $S' \to S | \epsilon, S \to aSb | ab$.

1. On introduit les non-terminaux A et B, on obtient la grammaire $S' \to S | \epsilon, S \to ASB | AB, A \to a, B \to b$.
2. on remplace la règle $S \to ASB$ par $S \to AC$ et $C \to SB$

2.3 La programmation dynamique

La programmation dynamique est utilisée pour résoudre des problèmes d'optimisation. La

conception d'un algorithme de programmation dynamique est typiquement découpée en trois étapes.

1. Caractériser la structure d'une solution.
2. Définir de manière récursive la solution.
3. Calculer la solution.

2.4 L'algo CYK et sa preuve.

Theorème 3 *(Algorithme de Cocke-Younger-Kasami) Le problème suivant admet un algorithme qui le résout en temps polynomial : Entrée : un mot w, une grammaire G sous forme normale de Chomsky Sortie : oui si $w \in L(G)$, non sinon*

Démonstration. Soit $w = w_1...w_n$ un mot donné en entrée du problème et G une grammaire sous FNC. On note $w[i,j]$ le mot $w_i \dots w_j$. On note $E_{i,j}$ l'ensemble des non-terminaux de G qui engendrent $w[i,j]$. On va alors calculer successivement les $E_{i,j}$ et on regardera si $S \in E_{1,n}$ (où S est l'axiome de la règle) pour répondre au problème posé.

Établissons d'abord une relation de récurrence : lorsqu'on considère une unique lettre de w, on a $A \in E_{i,i}$ si et seulement s'il existe une règle $A \to w_i$ dans G. Et pour le cas récursif, on a $A \in E_{i,j}$ (avec $i < j$) si et seulement s'il existe un $k \in [i; j-1]$ et une règle $A \to BC$ avec $B \in E_{i,k}$ et $C \in E_{k+1,j}$ (l'équivalence est permise par le fait que notre grammaire est donnée en forme normale de Chomsky). On obtient alors :

$\forall i \in [1;n], E_{i,i}$ = Non-terminaux A tel que $A \to w_i$ est une règle de G

$\forall i \in [1;n-1], \forall j \in [i+1;n], E_{i,j}$ = non terminaux A telle que $A \to BC$ est une règle de G, $B \in E_{i,k}, C \in E_{k+1,j}, i \leq k < j$

On en déduit immédiatement un algorithme de remplissage de E qui procède depuis la diagonale principale, diagonale après diagonale, vers le coin supérieur droit. L'algo fait varier trois indices i, j, k dans un intervalle de l'ordre de n. La complexité est en $O(n^3|G|)$ où $|G|$ est le nombre de règles qui constituent la grammaire G. Le problème initialement posé est donc bien résolu en temps polynomial en la taille de l'entrée.

La niche écologique de CYK. L'algorithme CYK est remarquable car il marche pour n'importe quelle grammaire hors-contexte, ambiguë ou pas. Cependant sa complexité cubique $O(n^3|G|)$ n'est pas suffisamment bonne pour la compilation, car n dépasse facilement 1000. On verra dans la suite d'autre algorithmes d'analyse avec une bien meilleure complexité (linéaire $O(n)$), mais qui ne marchent que pour certains types de grammaires hors contexte. Comme cela inclue les grammaires utilisées pour les langages de programmation, cette spécificité ne dérangera pas.

Trois exemples choisis. Une fois la matrice remplie, on peut redescendre depuis la case supérieure droite, vers les cases de la diagonale, pour voir s'inscrire dans la matrice un arbre de dérivation (ou plusieurs en cas de d'ambiguïté) du mot. Il faut pour cela se souvenir par où on est monté vers l'axiome. On fera cette manipulation pour les trois exemples.

1- $S \to abcd$ chomskysée. Interêt :c'est le plus simple que l'on puisse imaginer ; on récupère un joli petit arbre binaire en peigne, ou bien équilibré si on chomskyse de manière plus symétrique.

2- $S \to aSb, S \to ab$ chomskysée. Intérêt : il n'y a pas d'ambiguïté, et un seul arbre est possible. Il y a une grande régularité, on voit bien comment cet arbre s'inscrit dans la matrice.

3- $S \to SS, S \to a$ chomskysée. Intérêt : C'est le plus ambiguë que l'on puisse imaginer, La matrice est remplie de S, et cela décrit tout plein d'arbres.

3 Analyse syntaxique descendante

3.1 Grammaire facile pour l'analyse descendante.

Principe de l'analyse descendante : construire l'arbre de haut en bas, c'est-à-dire de la racine vers les feuilles. on réécrit toujours le non-terminal le plus à gauche En lisant le mot lettre par lettre on va savoir quel membre droit choisir lors de l'expansion du terminal le plus à gauche, pour construire l'arbre !

La grammaire facile en question : $G : S \to aSbT|cT|d \quad T \to aT|bS|c$ Nous allons analyser le mot $w = accbbadbc$

A chaque étape de l'analyse descendante, on a un symbole non-terminal courant à dériver (sur

l'arbre), et une lettre courante à engendrer (dans le mot à analyser).

Cette grammaire G est particulièrement adaptée à l'analyse descendante car, pour tout couple (N, x) où N est un non-terminal et x un terminal, au plus une suite de dérivations de N vers un mot commençant par x est possible. De plus, la première dérivation de cette suite est évidente à trouver car le membre droit de chaque règle commence toujours par un terminal.

3.2 Formalisme de l'analyse utilisant un automate à pile

On refait l'analyse précédente mais cette fois en utilisant un automate à pile. C'est seulement l'an prochain que l'on introduira sérieusement la notion formelle d'automate à pile, cependant on va quand même l'utiliser cette année pour mener l'analyse descendante, car c'est relativement facile. L'analyse se fait en utilisant une table avec trois colonnes : d'abord la pile, à gauche, ensuite le mot au centre (qui va être lu lettre par lettre) et finalement la règle utilisée a droite. Attention, il est important d'adopter la convention que "La pile tombe à gauche", i.e. le sommet de pile se trouve à gauche du mot qui représente la pile. Il y à deux actions possible pour notre automate à pile, en fonction de la nature du sommet de pile :

1. Si c'est un non-terminal il sera réécrit à l'étape suivante en choisissant le bon membre droit en fonction de la prochaine lettre du mot.
2. si c'est un terminal, alors cela doit être exactement la prochaine lettre du mot (sinon, le mot n'est pas reconnu), et on la dépile, en même temps qu'on avance sur le mot.

3.3 Table d'analyse.

Elle indique pour chaque non-terminal à expandre, vers quel membre droit expandre, en fonction de la prochaine lettre x du mot. Il y a donc les non-terminaux possibles en ordonnée, et les terminaux en abscisse, et des membres droits dans les cases. L'analyse va être déterministe s'il n'y a pas de choix ; i.e. au plus un membre droit possible par case. Pour notre grammaire facile, c'est facile : la case de coordonnée X, x contient le membre droit de la réécriture de X qui commence par x. Dans le cas général, on peut avoir des membres droits qui commencent par des non-terminaux, ou qui sont réduits à epsilon. Il va falloir que l'on passe par des calculs pour savoir au final par quoi vont commencer les mots que l'on va dériver, si on choisit tel ou tel membre droit.

3.4 Premier et Suivant

On définit deux sortes d'ensemble de lettres associés aux non-terminaux :
$premier(N \in \Sigma_N) = \{a \in \Sigma_T | \exists \beta \in \Sigma^\star, N \to a\beta\}$
$premier(\alpha \in \Sigma^\star) = \{a \in \Sigma^\star | \exists \beta \in \Sigma^\star, \alpha \to a\beta\}$
$suivant(N \in \Sigma_N) = \{a \in \Sigma | S \to \alpha N a \beta\}$

Ces ensembles se calculent en résolvant un système d'équations : Pour les premier, l'équation d'un non-terminal X s'écrit en regardant les règles qui réécrivent X, et en cherchant mentalement toutes les lettres qui commencent les membres droits de ces règles, ou qui commencent les mots dérivables à partir de ces membres droits. Ça fera des équations i.e. les premiers de X sont définis a partir des premiers de Y, Z ...si ces membres droits commencent par des non-terminaux Y, Z

Pour les suivants de X il faut regarder les occurrences de X dans les membres droits, en général on cherche à regarder encore une fois les occurrences dans les membres gauches et on se trompe. Les suivants sont les premiers des sous-mots qui suivent ces occurrences dans les membres DROITS. Et, de plus, si jamais ces occurrences se trouvent à la toute fin du membre droit, on va rajouter les suivants du non-terminal réécrit dans cette règle,i.e. le membre gauche. On obtient ainsi une fois de plus un système d'équations.

Dans les deux cas, le système s'écrit comme une équation $X = f(X)$ ou X est un vecteur d'ensemble de mots, et f est une fonction croissante sur ces vecteurs d'ensemble pour l'ordre suivant : un vecteur d'ensemble est plus petit qu'un autre si chacune de ses composantes est inclue dans la composante correspondante de l'autre vecteur. Le fait que f est croissante pour nos systèmes est en fait très simple : f est définie seulement à partir d'union, et l'union conserve l'inclusion, si A inclus dans A' et B inclus dans B', alors A union B est inclus dans A' union B'.

La résolution se fait en itérant la fonction f depuis l'élément minimum (le vecteur d'ensembles vides) jusqu'à obtention d'un point fixe. La suite

des itérations croit, dans un espace borné, donc ça va fatalement converger.

Exemple : On applique donc la méthode au calcul des suivants pour la grammaire : $E \to E+F|F, F \to F+G|G, G \to id|cte|(E)$ On rajoute $S \to E\#$ pour être sûr que tout le monde a un suivant.

On écrit la fonction f, on constate qu'elle est bien croissante, on résout l'équation par itération.

On obtient pour les premier, premier(E) = premier(F) ; premier(F) = premier(G) ; premier(G) = { '(', id, cte } ; premier (S) = premier (E).
Pour écrire le système je note X pour premier(X), j'utilise le même symbole du non-terminal pour dénoter l'ensemble de mots qu'on cherche ;
$f(E,F,G,S) = (F, G, \{'(', id, cte\}, E)$

Pour résoudre j'itère f en démarrant de (∅,∅,∅,∅)
on obtient donc la suite de vecteurs d'ensembles :
(∅,∅,∅,∅)
(∅,∅,{ '(', id, cte },∅)
(∅,{ '(', id, cte } ,{ '(', id, cte },∅)
({ '(', id, cte },{ '(', id, cte } ,{ '(', id, cte },∅)
({ '(', id, cte },{ '(', id, cte } ,{ '(', id, cte },{ '(', id, cte })

Après ça bouge plus, c'est le point fixe.

Trouver les équations pour les suivants est plus compliqué que pour les premiers, surtout en présence de non-terminaux "annulable", i.e. qui peuvent dériver ϵ : On exécute cet algorithme qui résume ce que j'ai déjà dit d'une autre manière plus systématique et formelle : Pour chaque non-terminal E, pour chaque occurrence de E dans un membre droit des productions, il faut regarder ce qui suit E :

- si c'est un terminal 'x', ajouter 'x' à Suivant(E)
- si c'est un non-terminal Y, ajouter Premier (Y) à Suivant(E)
- de plus si Y est annulable, reprendre à partir de ce qui suit Y
- si rien ne suit, ou (plus généralement) si tout ce qui suit est annulable, ajouter Suivant(A) à Suivant(E), où A est le membre gauche de la production.

Cet algorithme donne le système :
suivant(E) = { +, #, ')' } ;
suivant(F) = { + } union Suivant(E)
suivant(G) = Suivant(F)
suivant(S) = ∅
Avec les mêmes conventions de notation que pour premier, le système s'écrit
f(E,F,G,S) = ({ +, #, ')' } , { + } union E), F, ∅)
et l'itération donne :
(∅,∅,∅,∅)
({ +, #, ')' }, { + } ,∅,∅)
({ +, #, ')' },({ +, #, ')' } ,∅,∅)
({ +, #, ')' },({ +, #, ')' } ,{ +, #, ')' },∅)

3.5 Grammaires LL(1), cas général

On dira qu'une grammaire est $LL(k)$ si lire k caractères d'avance permet d'obtenir une table d'analyse descendante déterministe, i.e. au plus une possibilité pour chaque case. Dans la pratique on considère seulement le cas $k = 1$ et cela conduit au format de table d'analyse que nous avons déjà mis en place. On considère à présent des grammaires LL(1) plus compliquées que l'exemple simple qu'on a vu, qui mettent en jeu le calcul des premiers et des suivants.

Grammaire plus difficile avec membre droit commençant par des non-terminaux. Construire la table et analyser le fonctionnement de $G : S \to F|(S+F), \quad F \to 1$

Grammaire parmi les plus difficiles car contenant un membre droit nullifiable. On considère $S \to aSb|\epsilon$. ϵ n'a pas de premier alors que faire ? On choisira d'expandre S vers ϵ, si le prochain caractère du mot appartient à suivant(S), faire l'exemple, on comprend limpide ce qui se passe sur cette grammaire, du fait de son caractère minimalistique.

Remplissage de la table d'analyse. Il faut remplir chaque case de la table avec des membres droits.

1. Pour la famille des grammaires faciles, où les régles sont de la forme $X \to y\beta$, avec X non terminal, et y terminal, la case de coordonnées X, y recoit le membre droit $y\beta$
2. Pour les grammaires plus difficiles, avec des règles de la forme $X \to Y\beta$, avec X non terminal, et Y non-terminal, la ou les cases de

coordonnées X, y avec $y \in \text{prem}(Y)$ recoit le membre droit $Y\beta$

3. Pour les grammaires les plus difficiles, avec des règles de la forme $X \to \epsilon$, la ou les cases de coordonnée X, y avec $y \in \text{suiv}(X)$ recoit le membre droit ϵ

Ce petit synopsis n'est pas exact, il a été simplifié exprès pour rester digeste. Pour les grammaires plus difficiles, lorsque Y est nullifiable (il peut se réecrire en ϵ) il faut considérer les premiers de ce qui vient après. Pour les grammaires les plus difficiles, on va devoir calculer les suivants dans le cas plus général où le membre droit est nullifiable, car ne contenant que des non-terminaux nullifiables.