Chapitre 12

Parsage tabulaire

Nous nous étudions dans ce chapitre des stratégies d'analyse dédiées à l'analyse de langages ambigus, tels que ceux qui sont couramment utilisés pour décrire des énoncés en langage naturel. Rappelons qu'un langage CF ambigu est un langage tel que toute grammaire CF qui le représente est ambiguë, c'est-à-dire assigne plus plus d'un arbre de dérivation à au moins un mot de L. Dans ce contexte, les stratégies déterministes, garantissant une complexité linéaire, que nous avons présentées dans les chapitres précédents (notamment aux chapitres 7 et 9) sont nécessairement inadaptées. Le problème est double : (i) comment parvenir à conserver une complexité raisonnable pour tester l'appartenance d'une chaîne à un langage (ii) comment représenter efficacement l'ensemble (potentiellement exponentiellement grand) des différentes analyses pour une phrase donnée. Nous expliquons tout d'abord comment représenter dans une structure efficace, *la table des sous-chaînes bien formées*, un ensemble exponentiel d'arbres d'analyse. Nous présentons ensuite des algorithmes ayant une complexité polynomiale (précisément en $O(n^3)$, où n est la longueur de la chaîne à analyser) pour résoudre ce problème, tels que les algorithmes CYK (Younger, 1967) et Earley (Earley, 1970), qui sont présentés aux sections 12.1 et 12.2, ainsi que diverses variantes et extensions.

12.1 Analyser des langages ambigus avec CYK

12.1.1 Les grammaires pour le langage naturel

Nous avons vu (notamment au chapitre 6) que le parsage naïf de grammaires CF impliquait de calculer et de recalculer en permanence les mêmes constituants. Il apparaît donc naturel, pour accélérer la procédure, d'utiliser une structure temporaire pour garder les résultats déjà accumulés. Il existe, du point de vue du traitement automatique des langues , deux raisons importantes supplémentaires qui justifient le besoin de manipuler une telle structure. La première, et probablement la plus importante, est que le langage naturel est abominablement ambigu, et qu'en conséquence, les grammaires naturelles seront naturellement ambiguës, c'est-à-dire susceptibles d'assigner plusieurs structures syntaxiques différentes à un même énoncé. En sus de l'ambiguïté lexicale, massive, l'exemple le plus typique et potentiellement générateur d'une explosion combinatoire des analyses est celui du rattachement des groupes prépositionnels.

Considérons de nouveau la grammaire "des dimanches", reproduite à la Table 12.1, et intéressons-nous par exemple à la phrase : *Louis parle à la fille de la fille de sa tante*. Selon la manière dont on analyse le "rattachement" des groupes prépositionnels (correspondant au non-terminal *GNP*), on

p_1	S	$\rightarrow GN\ GV$	p_{15}	V	$\rightarrow$ *mange* | *sert*
p_2	GN	$\rightarrow DET\ N$	p_{16}	V	$\rightarrow$ *donne*
p_3	GN	$\rightarrow GN\ GNP$	p_{17}	V	$\rightarrow$ *boude* | *s'ennuie*
p_4	GN	$\rightarrow NP$	p_{18}	V	$\rightarrow$ *parle*
p_5	GV	$\rightarrow V$	p_{19}	V	$\rightarrow$ *coupe* | *avale*
p_6	GV	$\rightarrow V\ GN$	p_{20}	V	$\rightarrow$ *discute* | *gronde*
p_7	GV	$\rightarrow V\ GNP$	p_{21}	NP	$\rightarrow$ *Louis* | *Paul*
p_8	GV	$\rightarrow V\ GN\ GNP$	p_{22}	NP	$\rightarrow$ *Marie* | *Sophie*
p_9	GV	$\rightarrow V\ GNP\ GNP$	p_{23}	N	$\rightarrow$ *fille* | *cousine* | *tante*
p_{10}	GNP	$\rightarrow PP\ GN$	p_{24}	N	$\rightarrow$ *paternel* | *fils* |
p_{11}	PP	$\rightarrow$ *de* | *à*	p_{25}	N	$\rightarrow$ *viande* | *soupe* | *salade*
p_{12}	DET	$\rightarrow$ *la* | *le*	p_{26}	N	$\rightarrow$ *dessert* | *fromage* | *pain*
p_{13}	DET	$\rightarrow$ *sa* | *son*	p_{27}	ADJ	$\rightarrow$ *petit* | *gentil*
p_{14}	DET	$\rightarrow$ *un* | *une*	p_{28}	ADJ	$\rightarrow$ *petite* | *gentille*

TAB. 12.1 – La grammaire G_D des repas dominicaux

dispose pour cette phrase d'au moins trois analyses :

– un seul complément rattaché au verbe *parle* : *à la fille de la cousine de sa tante*
– deux compléments rattachés au verbe *parle* : *à la fille* ; *de la cousine de sa tante*
– deux compléments rattachés au verbe *parle* : *à la fille de la cousine* ; *de sa tante*

Notez qu'il y en aurait bien d'autres si l'on augmentait la grammaire, correspondant à la possibilité d'utiliser des groupes prépositionnels comme compléments circonstanciels (temps, lieu, moyen...), ainsi : *Ma sœur parle à la table d'à côté de la fille de la cousine de sa tante*.

Linguistiquement, la raison de la prolifération de ce type d'ambiguïté en français provient de la structure des groupes nominaux et noms composés qui sont majoritairement de la forme *N PP* (*DET*) *N*, éventuellement entrelardés ici où là de quelques adjectifs. Ceci est particulièrement manifeste dans les domaines techniques, où l'on parle de *termes* : *une machine de Turing*, *une turbine à vapeur*, *un réseau de neurones*, *un projet de développement d'un algorithme de résolution du problème de l'approximation d'équations aux dérivées partielles*....

Nous n'avons mentionné ici que des cas d'ambiguïtés globales, c'est-à-dire qui persistent au niveau de la phrase toute entière. Un analyseur perd également beaucoup de temps à explorer inutilement des ambiguïtés *locales*. Considérez par exemple la construction d'un nœud S dans une analyse ascendante de *l'élève de Jacques a réussi le contrôle*. Dans cet exemple, un analyseur naïf risque de construire toute la structure de racine S dominant le préfixe : *Jacques a réussi le contrôle*, qui, bien que correspondant à une phrase complète parfaitement correcte, laisse non analysé le préfixe *l'élève de*, et n'est donc d'aucune utilité.

Il est alors nécessaire, pour des applications en traitement des langues, de représenter de manière compacte un ensemble d'arbres de dérivations : on appelle un tel ensemble d'arbres une *forêt de dérivations*.

Second desiderata important : les systèmes de parsage envisagés jusqu'alors fournissent une réponse binaire (oui/non) à la question de l'appartenance d'une chaîne à un langage. Dans le cas où la réponse est négative, cela donne bien peu d'information aux traitements ultérieurs, quand bien même un grand nombre de constituants (incomplets) ont été formés pendant le parsage. Dans la mesure où les systèmes de TLN sont souvent confrontés à des énoncés non-grammaticaux au

sens strict (style de rédaction (eg. les titres) ; systèmes de dialogues : hésitations, reprises... ; sorties de systèmes de reconnaissance des formes (de la parole ou de l'écriture manuscrite) , cette manière de procéder est insatisfaisante : faute d'une analyse complète, on aimerait pouvoir produire au moins une analyse partielle de l'énoncé, qui pourra servir aux niveaux de traitement ultérieurs (par exemple l'analyseur sémantique).

La solution à tous ces problèmes consiste à utiliser une table des *sous-chaînes bien formées*, qui va d'une part mémoriser les structures partielles dès qu'elles ont été créées, afin d'éviter leur recalcul, et qui va, d'autre part, fournir des éléments d'informations aux niveaux de traitement supérieurs, même en cas d'échec de l'analyseur.

12.1.2 Table des sous-chaînes bien formées

La structure la plus généralement utilisée pour représenter des analyses partielles ou multiples est une table à double entrée : le premier indice, correspondant à la position de début du constituant, varie entre 0 et n ; le second indice correspond à la position du premier mot non couvert et varie entre 0 et $n + 1$. Chaque cellule $T[i, j]$ de cette table contient la liste des constituants reconnus entre i et j, i inclus, j exclus.

En supposant que la phrase à analyser soit : *ma sœur mange*, on aurait la table des sous-chaînes bien formée suivante (voir la Table 12.2).

4	S		V, GV
3	GN	N	
2	DET		
	ma	sœur	mange
	1	2	3

Tab. 12.2 – Table des sous-chaînes bien formées

La cellule $[1, 3]$ contient un GN, indiquant que l'on a reconnu un groupe nominal entre les positions 1 et 3 ; à l'inverse, la cellule $[2, 4]$ est vide : aucun constituant de longueur 2 ne commence à cette position.

Une manière alternative d'implanter cette table consiste à la représenter sous la forme d'un graphe orienté sans cycle (ou DAG pour *Directed Acyclic Graph*). Chaque nœud de ce graphe représente un mot (ou un indice de mot), et chaque transition entre nœuds est étiquetée par le constituant trouvé entre les deux nœuds. Plusieurs transitions entre deux nœuds représenteront alors des fragments ambigus, c'est-à-dire qui peuvent être analysés de plusieurs manières différentes.

Ce type de représentation de l'ambiguïté est illustrée par l'exemple de l'énoncé : *La fille parle à sa mère de sa tante*, dont la table des sous-chaînes bien formées est partiellement[1] représentée à la Figure 12.1. Nous avons volontairement introduit sur ce schéma une information supplémentaire, qui n'est pas représentée dans la table, en étiquetant chaque arc avec la production qui l'induit, alors la table n'enregistre en réalité que l'identité du non-terminal correspondant.

Comme figuré sur ce graphe, il existe deux manières d'analyser *à sa mère de sa tante* : soit comme un *GNP*, soit comme une succession de deux *GNP*s. Ces deux interprétations sont effectivement représentées dans la table, qui, pourtant, ne contient qu'un seul arc étiqueté GV entre les noeuds 3 et 10.

[1] Il manque donc des arcs : sauriez-vous dire lesquels ?

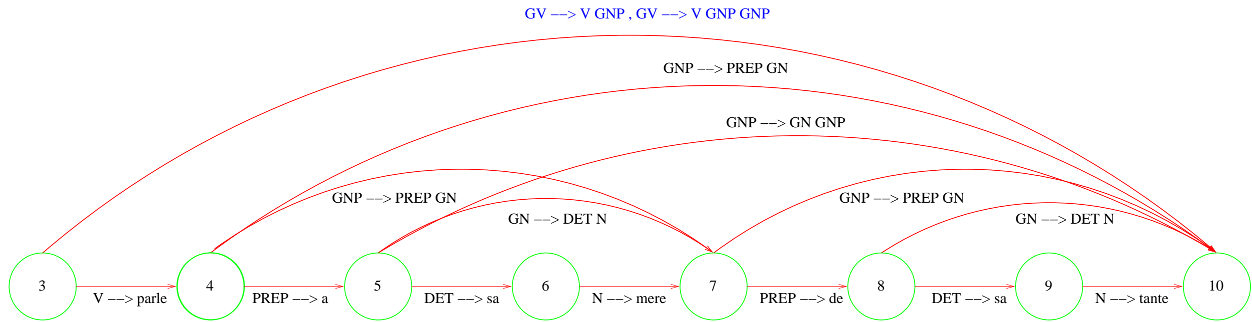

Fig. 12.1 – Représentation graphique d'une ambiguïté syntaxique

Dans cette représentation, le succès du parsage correspond à l'écriture dans la case "en haut à gauche" ($T[1, n]$) du tableau d'un constituant de type S (version tabulaire), ou encore d'un arc étiqueté S "couvrant" toute la phrase (version graphique).

Ces tables peuvent s'utiliser indépendamment de la stratégie de recherche ou de parsage utilisée, ascendante, descendante, mixte, dans la mesure où elles ne servent qu'à stocker de l'information. La seule contrainte d'utilisation est relative à l'ordre dans lequel le remplissage aura lieu. Ainsi, pour effectuer un parcours complet de l'espace de recherche, il faudra, dans une analyse descendante, vérifier que l'on introduit une nouvelle entrée dans la table qu'au moment où l'on connaît tous les constituants qui peuvent entrer dans cette case. Dans le cas contraire, on risque d'utiliser par anticipation, lors de la consultation de la table, un résultat partiel, et oublier des analyses. Le même type de vérification est nécessaire lorsque l'on adopte une stratégie de parsage ascendant.

12.1.3 L'algorithme CYK

L'algorithme CYK (Younger, 1967), du à Cocke, Younger et Kasami est destiné au parsage de grammaires sous forme normale de Chomsky (voir la section 8.2.1). Il fournit un premier exemple de d'utilisation de ces structures de table. Cet algorithme[2]implémente une stratégie d'analyse strictement ascendante. La mise sous CNF garantit en particulier, qu'on ne rencontrera ni production vide, ni chaîne de règles, deux configurations potentiellement problématiques pour des stratégies purement ascendantes. Commençons par donner une première version de cet analyseur, formalisé par l'algorithme 10.

La première étape de cet algorithme consiste à initialiser le tableau $T[]$ en insérant toutes les productions de type $A \rightarrow a$ potentiellement utilisées dans l'entrée u. Cette étape correspond au remplissage de la "diagonale" du tableau" ; elle est simplifiée par le fait que dans une grammaire sous forme normale de Chomsky, les seules règles introduisant des terminaux sont de la forme $A \rightarrow a$. La boucle principale consiste à construire, pour des tailles croissantes de l, les constituants de longueur l débutant en position i : il suffit, pour cela, de considérer toutes les factorisations en deux parties de $u_i \ldots u_{i+l}$ et de chercher celles qui sont couvertes par deux terminaux B et C, avec $A \rightarrow BC$ une production de la grammaire.

Notez que, même s'il existe deux règles $A \rightarrow BC$ et $A \rightarrow DE$ permettant d'insérer A dans la cellule $T[i, j]$, la cellule $T[i, j]$ ne contiendra qu'une seule occurrence de A. C'est cette "factorisation" qui permet de représenter un nombre potentiellement exponentiel d'analyses dans une table

[2]C'est plutôt d'un méta-algorithme qu'il faudrait en fait parler, car la dénomination CYK regroupe en fait des stratégies d'analyse variées : gauche droit, en ligne, non directionnel...

Algorithm 10 – Analyseur CYK pour une grammaire CNF $G = (\Sigma, V, S, P)$

// la phrase à analyser est $u = u_1 \dots u_n$
// la table d'analyse T est initialement vide : $T[i, j] = \emptyset$
for $j := 1 \dots n$ **do**
  foreach $A \to u_i \in P$ **do**
    $T[i, i+1] := T[i, i+1] \cup \{A\}$
  od
od
// Boucle principale : Cherche les constituants de longueur croissante
for $l := 2 \dots n$ **do**
  for $i := 1 \dots n - l + 1$ **do**
    for $k := i + 1 \dots i + l - 1$ **do**
      if $B \in T[i, k] \ \wedge \ C \in T[k, i + l] \ \wedge \ A \to BC \in P$
        then $T[i, k + l] := T[i, k + l] \cup \{A\}$
      fi
    od
  od
od
if $S \in T[1, n]$ **then** *return*(**true**) **else** *return*(**false**) **fi**

polynomiale.

Une fois la table construite, il suffit de tester si la table contient S dans la cellule $T[1, n]$ pour décider si la séquence d'entrée appartient ou non au langage engendré par la grammaire.

La correction de l'algorithme CYK repose sur *l'invariant* suivant, qui est maintenu de manière ascendante :

Proposition 12.1. *$A \in T[i, j])$ si et seulement si $A \overset{\star}{\Rightarrow}_G \ u_i \dots u_{j-1}$.*

La preuve de cette proposition repose sur une récurrence simple sur la longueur des constituants $(j - i)$. L'étape d'initialisation garantit que l'invariant est vrai pour les constituants de longueur 1. Supposons qu'il soit vrai pour tout longueur $\leq l$. L'insertion dans la cellule $T[i, j]$ est déclenchée uniquement par les situations où l'on a à la fois :
– $B \in T[i, k]$, ce qui implique que $B \overset{\star}{\Rightarrow} \ u_i \dots u_{k-1}$
– $C \in T[k, j]$, ce qui implique que $C \overset{\star}{\Rightarrow} \ u_k \dots u_{j-1}$
– $A \to BC \in P$, ce qui entraîne bien que $A \overset{\star}{\Rightarrow} \ u_i \dots u_{j-1}$

Quelle est la complexité de l'algorithme CYK ? La boucle principale du programme comprend trois boucles **for** se répétant au pire n fois, d'où une complexité cubique en n. Chaque cellule de T contient au plus $|V|$ valeurs ; dans le pire des cas, il y aura autant de productions CNF que de paires de non-terminaux, d'où une complexité $|V|^2$ pour la recherche de la partie droite. Au total, l'algorithme CYK a donc une complexité totale : $|V|^2 n^3$. C'est un premier résultat important : dans le pire des cas, le test d'appartenance de u à une grammaire CF G a une complexité polynomiale en la longueur de u, bornée par un polynôme de degré 3. Notez que la complexité dépend, à travers la constante, de la taille de grammaire (après Chomsky normalisation).

Différentes variantes de CYK sont possibles, selon que l'on adopte un ordre de parcours qui est

soit « les plus courtes chaînes d'abord » (ce qui correspond à l'algorithme présenté ci-dessus), soit « par sous-diagonales ». Ces deux parcours sont représentés respectivement en trait pleins et pointillés dans la Figure 12.2. Cette table illustre le fait que, dans le premier parcours, la cellule $T[1,4]$ est traitée après la cellule $T[5,7]$), c'est-à-dire après l'examen de toutes les sous-chaînes de longueur de 2 ; alors que dans le second parcours, elle est examinée dès que deux sous-constituants sont disponibles. Cette seconde stratégie correspond à une analyse *en ligne*, pendant laquelle les constituants maximaux sont construits au fur et à mesure que les mots sont présentés à l'analyseur.

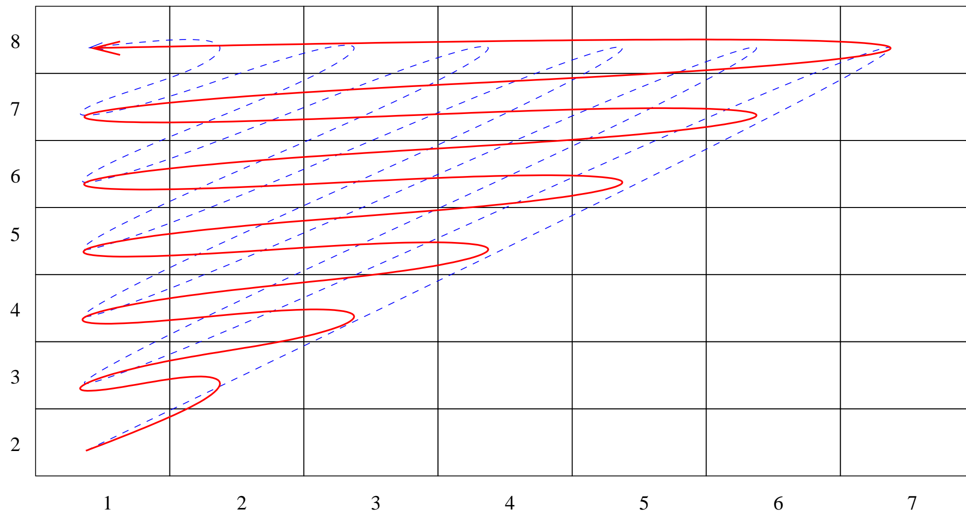

FIG. 12.2 – Deux implantations différentes de CYK

Pour compléter, notez que CYK est adaptable aux cas non-CNF et qu'il est possible, en analysant la version Chomsky-normalisée, de reconstruire (au prix d'une structure auxiliaire gérant la correspondance entre règles CNF et les autres) les analyses fournies par la grammaire originale.

12.1.4 Du test d'appartenance à l'analyse

Comment reconstruire à partir de la table T les différents arbres de dérivations ? Cette étape de reconstruction, implantée naïvement, demande deux extensions à l'algorithme précédent 10. Il s'agit tout d'abord de sauvegarder la manière dont un non-terminal est inséré dans une cellule, en associant à chaque non-terminal A de $T[i,j]$ un ensemble BP, contenant par exemple des quintuplets[3] (r,i,j,i',j'), où r est l'indice de la A-production appliquée, et (i,j) et (i',j') les indices dans la table des "descendants" de A.

Avec cette extension, la construction de l'arbre d'analyse s'opère en traversant depuis la racine le graphe de descendance induit par les BP. La procédure *MakeTrees* de l'algorithme 11 implémente un tel parcours.

[3]Si l'on y réfléchit un peu, des quadruplets suffisent. Pourquoi ?

Algorithm 11 – Construction des arbres de dérivation à partir d'une table CYK

```
Tree = ∅ // l'arbre est une pile contenant les dérivations gauches
MakeTrees(S, 1, n)
funct MakeTrees(A, p, l))
  if (l = 1)
    then do // une feuille
          foreach (A → a) ∈ BP(A, p, l) do
            push(Tree, A → a)
            if s = n then PrintTree(Tree) fi
          od
       od
    else do // un noeud interne
          foreach (A → BC, i, j, i', j') ∈ BP(A, l, s)
            MakeTrees(B, i, j)
            MakeTrees(C, i', j')
          od
       od
  fi
```

12.2 Algorithme d'Earley et ses variantes

12.2.1 Table active des sous-chaînes bien formées

Pour l'instant, nous avons considéré la table comme un simple auxiliaire pour mémoriser les constituants trouvés lors de la recherche et pour représenter des analyses partielles et/ou multiples. Nous allons maintenant montrer comment étendre cette représentation pour *diriger* l'analyse, en utilisant la table pour représenter des hypothèses concernant les constituants en cours d'agrégation (dans une stratégie ascendante) ou concernant les buts à atteindre (dans le cas d'une stratégie descendante). Cela nous permettra (i) de nous affranchir de la précondition de mise sous forme normale de Chomsky et (ii) de construire des analyseurs plus efficaces.

Un premier pas, par rapport à CYK, consiste à stocker dans la table d'analyse un peu plus que les simples terminaux reconnus entre deux positions, en utilisant de nouveau (cf. la section 7.2) le concept de *règle pointée* pour représenter les hypothèses et buts courants. Rappelons :

Définition 12.1 (Règle pointée). *Une* règle pointée *est une production augmentée d'un point. Le point indique l'état courant d'application de la règle (autrement dit à quel point les hypothèses qui permettent d'appliquer la règle ont été satisfaites).*

Par exemple, des règles pointées suivantes sont tout à fait licites compte-tenu de la Table 12.1 :
– $S \rightarrow \bullet\ GN\ GV$
– $S \rightarrow GN \bullet GV$
– $S \rightarrow GN\ GV \bullet$
Ces arcs s'insèrent naturellement dans la table des sous-chaînes bien formées, comme représenté à la Figure 12.3. Sur cette figure, la règle pointée $S \rightarrow GN \bullet GV$ entre les nœuds 1 et 3 indique que l'opération réussie de réduction du GN a conduit à reconnaître le début d'une phrase (S), à laquelle il manque encore un groupe verbal pour être complète.

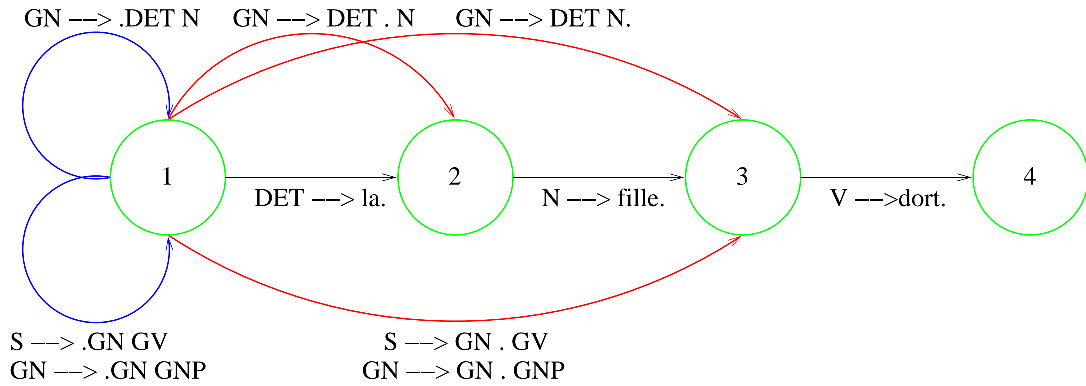

FIG. 12.3 – Un exemple de table active

Le graphe de la Figure 12.3 se transpose directement dans la représentation tabulaire suivante (voir la Table 12.3).

4	$S \rightarrow GN\ GV\bullet$		$V \rightarrow dort\bullet$
3	$S \rightarrow GN \bullet GV, GN \rightarrow GN \bullet GNP$	$N \rightarrow fille\bullet$	
2	$DET \rightarrow la\bullet, GN \rightarrow DET \bullet N$		
	1 la	2 fille	3 dort

TAB. 12.3 – Une table d'analyse en construction

On appellera *item* tout élément figurant dans une cellule de la table d'analyse. Dans le cas présent, un item est de la forme $[A \rightarrow \alpha \bullet \beta, i, j]$, portant l'information suivante : $\alpha \overset{\star}{\Rightarrow}_G \; u_i \ldots u_{j-1}$. Il existe en fait deux sortes d'items dans une table d'analyse :

Définition 12.2 (Items actifs et inactifs). *Un item actif est un item tel que le point n'est pas situé à droite de la partie droite. Un item inactif est un item qui n'est pas actif. Une table active est une table qui contient des items actifs.*

Dans l'exemple de la Table 12.3, $[S \rightarrow GN \bullet GV, 1, 3]$ est un item actif, alors que $[N \rightarrow fille\bullet, 2, 3]$ est inactif. Les items *inactifs* correspondent précisément à ce qui est représenté (et représentable) dans la table d'un analyseur de type CYK. Voyons comment les items actifs interviennent pour définir des stratégies de parsage plus efficaces, en commençant par une nouvelle stratégie purement ascendante.

12.2.2 La règle fondamentale du parsage tabulaire

L'idée générale du parsage à base de table, consiste à essayer d'étendre les items actifs pour les rendre inactifs, puisqu'une analyse complète est par définition représentée par un item inactif $[S \rightarrow \alpha\bullet, 1, n]$. Comment cette opération essentielle se déroule-t-elle ? Par application de la règle suivante :

Définition 12.3 (Règle fondamentale du parsage tabulaire). *Si la table d'analyse T contient l'item actif* $[A \rightarrow \alpha_1 \bullet B\alpha_2, i, j]$ *et l'item inactif* $[B \rightarrow \alpha_3\bullet, j, k]$*, alors on peut rajouter dans la table l'item (actif ou inactif) :* $[A \rightarrow \alpha_1 B \bullet \alpha_2, i, k]$.

Cette règle, par la suite dénotée *comp*, peut être vue comme une règle de déduction permettant de construire de nouveaux items en combinant des items déjà présents dans la table.

Un exemple d'application de cette règle fondamentale se lit dans l'exemple la Table 12.3 : ainsi la présence de l'item $[GN \rightarrow DET \bullet N, 1, 2]$ (un GN en cours de reconnaissance), et de l'item $[N \rightarrow fille \bullet, 2, 3]$ (un N complet) permet d'insérer dans la table le nouvel item $[GN \rightarrow DET N \bullet, 1, 3]$ (un GN complet, de longueur 2, débutant en position 1).

Pour achever de définir complètement un algorithme de parsage utilisant ces items, trois points supplémentaires doivent être spécifiés :
– l'initialisation de la table. En effet, la règle fondamentale ne pourra rien dériver tant que la table est vide : il faut donc décider d'une certaine manière initialiser la table.
– la stratégie d'utilisation des règles (ou comment insérer de nouveaux arcs actifs dans le graphe) : il faut également des arcs actifs pour appliquer la règle fondamentale ;
– la stratégie de recherche, qui correspond à la définition d'un ordre pour examiner efficacement les hypothèses actives.

Ces spécifications complémentaires doivent être penser dans l'optique permettre d'optimiser l'analyse, sachant que performances des algorithmes de parsage tabulaire dépendront principalement :
– du nombre d'items créés en cas d'échec de l'analyse : on souhaite qu'il y en ait le moins possible.
– du nombre d'items inutiles en cas de succès de l'analyse : idéalement tous les arcs du graphes doivent être utilisés dans au moins une analyse ; idéalement, il ne devrait plus y avoir d'arcs actifs quand l'analyse se termine.

Nous présentons, dans un premier temps, une instanciation particulière du parsage tabulaire dans le cadre d'une stratégie gauche droite purement ascendante. Nous présentons ensuite diverses variantes, plus efficaces, d'algorithmes utilisant la tabulation.

12.2.3 Parsage tabulaire ascendant

L'implémentation de cette stratégie se fait directement, en complétant la règle fondamentale par les instructions suivantes :
– *init* la règle fondamentale nécessite de disposer d'items actifs. La manière la plus simple pour en construire est d'insérer, pour chaque production $A \rightarrow \alpha$ de P, et pour chaque position dans la phrase, une hypothèse concernant le début de la reconnaissance de A. Comme ces nouveaux items ne "couvrent" aucun terminal de l'entrée courante, leur indice de début sera conventionnellement pris égal à l'indice de fin. Cette étape donne donc lieu à des items $[A \rightarrow \bullet\alpha, i, i]$ selon :

  foreach $(A \rightarrow \alpha) \in P$ **do**
    foreach $i \in (1 \ldots n)$ **do**
      insert$[A \rightarrow \bullet\alpha, i, i]$ in T
    od
  od

– *scan* il est également nécessaire de définir une stratégie pour concernant la reconnaissance des terminaux. La règle suivante exprime le fait que si un item actif "attend" un terminal, alors cet élément trouvé sur l'entrée courante (u) permettra le développement de l'item, soit formellement :

  if $[A \rightarrow \alpha \bullet a\beta, i, j] \in T \wedge u_j = a$
    then *insert*$[A \rightarrow \alpha a \bullet \beta, i, j+1]$ in T
  fi

La mise en œuvre de cette stratégie sur l'exemple de la phrase : *un père gronde sa fille* est illustrée à la Table 12.4. Cette table a été délibérément expurgée des (très nombreux) items créés à

l'initialisation, qui sont néanmoins utilisés pour dériver de nouveaux items.

Num.	Item	règle	Antécédents
1	$[DET \rightarrow un\bullet, 1, 2]$	*scan*	$[DET \rightarrow \bullet un, 1, 1]$
2	$[GN \rightarrow DET \bullet N, 1, 2]$	*comp*	$[GN \rightarrow \bullet DET\ N, 1, 1]$ et 1
3	$[N \rightarrow père\bullet, 2, 3]$	*scan*	$[N \rightarrow \bullet père, 2, 2]$
4	$[GN \rightarrow DET\ N\bullet, 1, 3]$	*comp*	2 et 3
5	$[S \rightarrow GN \bullet GV, 1, 3]$	*comp*	$[S \rightarrow \bullet GN\ GV, 1, 1]$ et 4
6	$[GN \rightarrow GN \bullet GNP, 1, 3]$	*comp*	$[GN \rightarrow \bullet GN\ GNP, 1, 1]$ et 4
7	$[V \rightarrow gronde\bullet, 3, 4]$	*scan*	$[V \rightarrow \bullet gronde, 3, 3]$
8	$[GV \rightarrow V\bullet, 3, 4]$	*comp*	$[GV \rightarrow \bullet V, 3, 3]$ et 7
9	$[GV \rightarrow V \bullet GN, 3, 4]$	*comp*	$[GV \rightarrow \bullet V\ GN, 3, 3]$ et 7
10	$[GV \rightarrow V \bullet GNP, 3, 4]$	*comp*	$[GV \rightarrow \bullet V\ GNP, 3, 3]$ et 7
11	$[GV \rightarrow V \bullet GN\ GNP, 3, 4]$	*comp*	$[GV \rightarrow \bullet V\ GN\ GNP, 3, 3]$ et 7
12	$[GV \rightarrow V \bullet GNP\ GNP, 3, 4]$	*comp*	$[GV \rightarrow \bullet V\ GNP\ GNP, 3, 3]$ et 7
13	$[S \rightarrow GNGV\bullet, 1, 4]$	*comp*	5 et 8
14	$[DET \rightarrow sa\bullet, 4, 5]$	*scan*	$[DET \rightarrow \bullet sa, 4, 4]$
15	$[GN \rightarrow DET \bullet N, 4, 5]$	*comp*	$[GN \rightarrow \bullet DET\ N, 4, 4]$ et 14
16	$[N \rightarrow fille\bullet, 5, 6]$	*scan*	$[N \rightarrow \bullet fille, 5, 5]$
17	$[GN \rightarrow DET\ N\bullet, 4, 6]$	*comp*	14 et 16
18	$[GV \rightarrow V\ GN\bullet, 3, 6]$	*comp*	9 et 17
19	$[GV \rightarrow V\ GN\ GNP\bullet, 3, 6]$	*comp*	11 et 17
20	$[S \rightarrow GNGV\bullet, 5, 1]$	*comp*	6 et 18
21	$[S \rightarrow GN \bullet GV, 4, 6]$	*comp*	$[S \rightarrow \bullet GN\ GV, 4, 4]$ et 17
22	$[GN \rightarrow GN \bullet GNP, 4, 6]$	*comp*	$[GN \rightarrow \bullet GN\ GNP, 4, 4]$ et 17

TAB. 12.4 – Parsage tabulaire ascendant

De manière implicite, le remplissage de la Table 12.4 obéit à une stratégie d'application des règles suivant :
– insérer tous les items résultant de l'application de *init*
– s'il est possible d'appliquer *comp*, choisir d'appliquer *comp*
– sinon s'il est possible d'appliquer *scan*, appliquer *scan*
Indépendamment d'une choix d'une stratégie de prioritisation des règles, l'implantation d'un algorithme demande de maintenir dans des structures séparées les items avant et après que l'on a évaluer leurs « conséquents ». Dans le jargon du domaine, ces deux structures sont la table (ou *chart*) et *l'agenda*. De manière très schématique, l'algorithme précédent se récrit en
– insérer tous les items résultant de l'application de *init* dans l'agenda.
– prendre un item de l'agenda, l'insérer dans la table, et insérer tous ses conséquents directs dans l'agenda.
Resterait encore pour achever la spécification complète d'un algorithme à définir comment choisir entre plusieurs applications concurrentes de la même règle (ainsi la rafale de *comp* donnant lieu aux items 8 à 13). Ainsi qu'à mieux analyser l'influence des malformations possibles de la grammaire (productions vides, productions non-génératives, récursions gauches...).

Une analyse plus poussée de cet algorithme révèle un certain nombre d'inefficacités :

1. *init* met à disposition de *comp* tous les éléments potentiellement utiles pour faire de l'analyse ascendante. Cette étape est toutefois bien trop prolixe et conduit à l'insertion dans la table de nombreux items qui ne seront jamais considérés ; le cas le plus flagrant étant l'insertion d'items actifs introduisant des terminaux qui n'apparaissent même pas dans la phrase.

2. *comp* est parfois utilisée inutilement : ainsi la création de l'item (inutile) 13, qui ne pourra jamais être développé puisque S n'apparaît dans aucune partie droite.

   De même, 21 et 22 sont inutiles, 21 doublement d'ailleurs parce que (i) le GV manquant ne pourra pas être trouvé au delà de la position 6 et (ii) quand bien même il le serait, reconnaître un S en position 4 ne pourra que conduire l'analyse dans une impasse.

Nous examinons, dans les variantes décrites ci-dessous, divers remèdes à ces inefficacités.

12.2.4 Coin gauche

L'approche "coin gauche" consiste essentiellement à revenir à une stratégie initialisation de la table semblable à celle mise en oeuvre pour CYK. Cela suppose que tous les terminaux soient introduits par des règles de la forme $A \rightarrow a$ et que ces règles soient les seules à introduire des terminaux.[4]Considérons que c'est bien le cas, comme dans la Table 12.1, ce qui nous autorise alors à remplacer l'étape *init* par une nouvelle définition selon :

for $i \in (1 \dots n)$ **do**
  foreach $A \rightarrow u_i$
    $insert([A \rightarrow u_i \bullet, i, i+1])$ in T
  od
od

Le problème est maintenant de déclencher l'insertion d'items actifs, qui permettront de nourrir la règle fondamentale (*comp*). L'idée de l'analyse du "coin gauche" consiste à émettre une hypothèse sur la reconnaissance d'un constituant que si l'on a déjà complètement achevé de reconnaître son "coin gauche" (c'est-à-dire le premier élément, terminal ou non terminal, de la partie droite). Cette stratégie se formalise par la règle *leftc* :

if $[Y \rightarrow \alpha \bullet, i, j] \in T$
 then **foreach** $X \rightarrow \beta Y \beta \in P$ **do**
    $insert([X \rightarrow Y \bullet \beta], i, j)$ in T
   od
fi

Cette nouvelle règle effectue en un coup la reconnaissance du coin gauche à partir d'un item inactif, là où l'algorithme précédent décomposait cette reconnaissance en une étape *init* (créant un item actif initial) et une étape de *comp* (reconnaissant le coin gauche).

En présence de productions ε, il faudrait également se donner la possibilité de reconnaître des productions vides par une règle de type :

foreach $A \rightarrow \varepsilon \in P$ **do**
 foreach $i \in 1 \dots n)$ **do**
  $insert([A \rightarrow \bullet \beta], i, i)$ in T
 od
od

[4]Ces deux conditions sont de pure convenance : on pourrait aussi bien définir un analyseur du coin gauche pour une grammaire ayant une forme quelconque et traiter les terminaux au moyen de deux règles : l'une effectuant un *scan* spécifique pour les terminaux correspondant à des coins gauches, et l'autre effectuant un *scan* pour les terminaux insérés dans les queues de parties droites. Sauriez-vous mettre en oeuvre une telle analyse ?

Il suffit maintenant, pour compléter cette description, de spécifier un ordre d'application de ces différentes règles. Les ordonner selon : *comp* est prioritaire sur *leftc*, elle-même prioritaire sur *init*, conduit à la trace suivante (voir la Table 12.5). Notons qu'ici encore le choix d'appliquer *leftc* avant d'en avoir fini avec les appels à *init* n'est pas une nécessité et un autre ordre aurait en fait été possible.

Num.	Item	règle	Antécédents
1	$[DET \rightarrow un\bullet, 1, 2]$	*init*	
2	$[GN \rightarrow DET \bullet N, 1, 2]$	*leftc*	1
3	$[N \rightarrow père\bullet, 2, 3]$	*init*	
4	$[GN \rightarrow DET\ N\bullet, 1, 3]$	*comp*	2 et 3
5	$[S \rightarrow GN \bullet GV, 1, 3]$	*leftc*	4
6	$[GN \rightarrow GN \bullet GNP, 1, 3]$	*leftc*	4
7	$[V \rightarrow gronde\bullet, 3, 4]$	*init*	
8	$[GV \rightarrow V\bullet, 3, 4]$	*leftc*	7
9	$[S \rightarrow GN\ GV\bullet, 1, 4]$	*compl*	5 et 8
10	$[GV \rightarrow V \bullet GN, 3, 4]$	*leftc*	7
11	$[GV \rightarrow V \bullet GNP, 3, 4]$	*leftc*	7
12	$[GV \rightarrow V \bullet GN\ GNP, 3, 4]$	*leftc*	7
13	$[GV \rightarrow V \bullet GNP\ GNP, 3, 4]$	*leftc*	7
14	$[DET \rightarrow sa\bullet, 4, 5]$	*init*	
15	$[GN \rightarrow DET \bullet N, 4, 5]$	*leftc*	14
16	$[N \rightarrow fille\bullet, 5, 6]$	*init*	
17	$[GN \rightarrow DET\ N\bullet, 4, 6]$	*compl*	15 et 16
18	$[GV \rightarrow V\ GN\bullet, 3, 6]$	*comp*	10 et 17
19	$[GV \rightarrow V\ GN \bullet GNP, 3, 6]$	*comp*	12 et 17
20	$[S \rightarrow GN\ GV\bullet, 1, 6]$	*comp*	6 et 18
21	$[S \rightarrow GN \bullet GV, 4, 6]$	*leftc*	17
22	$[GN \rightarrow GN \bullet GNP, 4, 6]$	*leftc*	17

Tab. 12.5 – Parsage tabulaire : stratégie du coin gauche

La différence avec la trace précédente (cf. la Table 12.4) ne saute pas aux yeux ; elle est pourtant sensible puisque les 22 items de la Table 12.5 sont les *seuls items construits*, alors qu'il fallait, pour avoir la liste complète des items de la table précédente, y ajouter les $n \times |P|$ items initiaux. Remarquons également la présence d'un certain nombre d'items inutiles dans la Table 12.5 : ainsi les items 6, 8, 9, 19, 21 et 22, soit presqu'un item sur trois.

12.2.5 Une stratégie mixte : l'algorithme d'Earley

Répétons-nous : l'efficacité d'un système de parsage dépend fortement du nombre d'items qui ont été inutilement ajoutés dans la table (et corrélativement développés). Nous l'avons mentionné plus haut, dans le cas d'un algorithme purement ascendant, l'ajout d'items inutiles est en fait chose courante. Considérez, en effet, l'analyse d'une phrase comme : *le fils de ma tante pleure*. La reconnaissance du groupe nominal (*GN*) *ma tante*, entraîne l'insertion ascendante des hypothèses correspondant à la règle $S \rightarrow GN\ GV$. Ces hypothèses vont, de plus, être prolongées, puisqu'on va bien trouver par la suite le *GN* (*ma tante*) comme le GV (*pleure*). Pourtant, il est impossible de trouver ici une phrase, puisqu'on laisserait inanalysés les trois premiers mots de la phrase (*le fils de*).

L'idée de l'algorithme d'Earley (Earley, 1970) est de guider la stratégie purement ascendante décrite à la section 12.2.3 par des informations de contrôle descendantes, qui vont permettre de ne développer que les items qui peuvent intervenir dans une analyse complète. À cet effet, la table est complétée par des items correspondant à des *prédictions* (descendantes) ; un hypothèse ne sera alors développée que si elle est correspond à une telle prédiction.

Cette idée est mise en œuvre en remplaçant l'étape d'initialisation aveugle de l'algorithme purement ascendant par une initialisation plus raisonnable, consistant à faire l'hypothèse minimale que l'on débute la reconnaissance d'une phrase (S) à la position 1. Ceci est formalisé par une nouvelle version de *init* :

foreach $S \rightarrow \alpha$ **do**
  insert($[S \rightarrow \bullet\alpha, 1, 1]$) in T
od

La mise en œuvre de cette approche demande une nouvelle règle exprimant les prédictions descendantes : il s'agit d'exprimer le fait qu'on ne fait l'hypothèse de démarrage d'un nouveau constituant que si ce constituant est attendu, ie. que si sa reconnaissance pourra être utilisée pour développer un item déjà existant. Cette nouvelle règle (*pred*) s'écrit donc[5] :

if $[X \rightarrow \alpha \bullet B\beta, i, j] \in T$
  then *insert*($[B \rightarrow \bullet\alpha, j, j]$) in T
fi

L'idée générale de l'algorithme est alors la suivante :
– initialiser les items actifs correspondants à S avec *init*
– appliquer les trois règles restantes, avec l'ordre de priorité suivant : *comp* est prioritaire sur *pred*, lui-même prioritaire sur *scan*.

La Table 12.6 donne la trace de l'utilisation de cette stratégie sur la phrase *Le fils de ma tante pleure*. Conformément aux idées originales de Earley, nous nous sommes permis une petite licence consistant à effectuer un regard avant de 1 au moment de chercher les terminaux. Ceci économise tout un tas de prédictions inutiles d'items tels que $[DET \rightarrow \bullet ma]$... Un résultat identique serait obtenu en complétant l'initialisation par un remplissage "à la CYK" des items introduisant des terminaux. Ce scan un peu amélioré est noté *scan+*.

L'incorporation du contrôle descendant des hypothèses permet donc effectivement d'éviter de construire un certain nombre d'items, comme en particulier ceux qui correspondraient à une hypothèse de démarrage de phrase au milieu (c'est-à-dire après la reconnaissance du *GN ma tante*) de l'énoncé. Sous cette forme, il apparaît toutefois que de nombreux items inutiles sont crées, soit parce que le mécanisme de prédiction est trop libéral (ainsi les items 10 à 14, 28 à 33 (seul 29 est utile), ou ceux créés après l'item 40 ; soit parce que la complétion agit sur des items sans avenir (ainsi les items 36 à 39).

Extensions Il est donc souhaitable et possible de raffiner encore cet algorithme :
– en incorporant des possibilités de regard avant (*look-ahead*), qui vont permettre, à nouveau, d'éviter de développer certains items. Ainsi par exemple, l'item 25 est inséré par prédiction d'un GNP. Or, il suffirait de regarder l'input pour réaliser que c'est impossible, puisqu'un GNP ne peut démarrer qu'avec une des prépositions, et que l'input contient un verbe. Comme pour les analyseurs LL, ce type de filtrage demanderait de calculer la table des *FIRST*. Il est également

[5] Une manière d'interpréter cette règle est de la voir comme simulant, de concert avec *comp*, la construction dynamique d'une table d'analyse LR (voir la section 7.2).

1	$[S \rightarrow \bullet GN\ GV, 1, 1])$	*init*	
2	$[GN \rightarrow \bullet DET\ N, 1, 1])$	*pred*	1
3	$[GN \rightarrow \bullet GN\ GNP, 1, 1])$	*pred*	1
4	$[DET \rightarrow le\bullet, 1, 2])$	*scan+*	2
5	$[GN \rightarrow DET \bullet N, 1, 2])$	*comp*	1,4
6	$[N \rightarrow fils\bullet, 2, 3])$	*scan+*	5
7	$[GN \rightarrow DETN\bullet, 1, 3])$	*comp*	2, 6
8	$[S \rightarrow GN \bullet GV, 1, 3])$	*comp*	1, 7
9	$[GN \rightarrow GN \bullet GNP, 1, 3])$	*comp*	3,7
10	$[GV \rightarrow \bullet V, 3, 3]$	*pred*	8
11	$[GV \rightarrow \bullet V\ GN, 3, 3]$	*pred*	8
12	$[GV \rightarrow \bullet VGNP, 3, 3]$	*pred*	8
13	$[GV \rightarrow \bullet VGNGNP, 3, 3]$	*pred*	8
14	$[GV \rightarrow \bullet V\ GNP\ GNP, 3, 3]$	*pred*	8
15	$[GNP \rightarrow \bullet PP\ GN, 3, 3]$	*pred*	9
16	$[PP \rightarrow de\bullet, 3, 4])$	*scan+*	15
17	$[GNP \rightarrow PP \bullet GN, 3, 4]$	*comp*	15 et 16
18	$[GN \rightarrow \bullet DET\ N, 4, 4])$	*pred*	17
19	$[GN \rightarrow \bullet GN\ GNP, 4, 4])$	*pred*	17
20	$[DET \rightarrow ma\bullet, 4, 5])$	*scan+*	18
21	$[GN \rightarrow DET \bullet N, 4, 5])$	*comp*	18 et 20
22	$[N \rightarrow tante\bullet, 5, 6])$	*scan+*	21
23	$[GN \rightarrow DET\ N\bullet, 4, 6])$	*comp*	21 et 22
24	$[GNP \rightarrow PP\ GN\bullet, 3, 6]$	*comp*	17 et 23
25	$[GN \rightarrow GN\ GNP\bullet, 1, 6])$	*comp*	24 et 9
26	$[S \rightarrow GN \bullet GV, 1, 6])$	*comp*	1, 25
27	$[GN \rightarrow GN \bullet GNP, 1, 6])$	*comp*	3,7
28	$[GV \rightarrow \bullet V, 6, 6]$	*pred*	26
29	$[GV \rightarrow \bullet V\ GN, 6, 6]$	*pred*	26
30	$[GV \rightarrow \bullet VGNP, 6, 6]$	*pred*	26
31	$[GV \rightarrow \bullet VGNGNP, 6, 6]$	*pred*	26
32	$[GV \rightarrow \bullet V\ GNP\ GNP, 6, 6]$	*pred*	26
33	$[GNP \rightarrow \bullet PP\ GN, 6, 6]$	*pred*	27
34	$[V \rightarrow pleure\bullet, 6, 7])$	*scan+*	33
35	$[GV \rightarrow V\bullet, 6, 7]$	*comp*	34
36	$[GV \rightarrow V \bullet GN, 6, 7]$	*comp*	34
37	$[GV \rightarrow V \bullet GNP, 6, 7]$	*comp*	34
38	$[GV \rightarrow V \bullet GNGNP, 6, 7]$	*comp*	34
39	$[GV \rightarrow V \bullet GNP\ GNP, 6, 7]$	*comp*	34
40	$[S \rightarrow GN\ GV\bullet, 1, 7])$	*comp*	26 et 35
41	$[GN \rightarrow \bullet DET\ 7, 7]$	*pred*	36
42	$[GNP \rightarrow \bullet PP\ GN, 7, 7]$	*pred*	37
	...	*pred*	41

TAB. 12.6 – Développement de la table d'analyse avec l'analyseur d'Earley

possible de filtrer des complétions abusives par utilisation d'un regard avant : de cette manière les items 36 à 39 pourraient-ils être également évités. Le bénéfice de ces stratégies est toutefois contesté, dans la mesure où elles induisent un coût algorithmique non négligeable.
– en retardant les prédictions jusqu'à la reconnaissance d'un coin gauche. Ceci peut se voir comme une extension relativement naturelle de l'algorithme de la section 12.2.4, consistant à :
  – ajouter un item supplémentaire "descendant" à l'initialisation, de la forme $[S \rightarrow \bullet\alpha, 1, 1]$
  – ajouter une condition supplémentaire à la règle du coin gauche, demandant qu'un item $[A \rightarrow B \bullet \alpha, i, j]$ ne soit inséré que (i) si $[B \rightarrow \bullet\beta, i, j]$ existe (coin gauche normal) et si (ii) A est prédit de manière descendante, correspondant à un item $[X \rightarrow \gamma \bullet Y\delta, l, i]$, et A peut débuter une dérivation de Y.

Complexité L'algorithme d'Earley permet de construire des analyseurs pour des grammaires quelconques , avec une complexité également $O(n^3)$: l'analyseur doit considérer un par un tous les items de l'agenda pour les insérer dans la table. Le nombre de productions pointées étant une constante de la grammaire, il y a au maximum de l'ordre de $O(n^2)$ items ; l'examen d'un item $[A \rightarrow \alpha \bullet B\ \beta, i, j]$, notamment l'étape de complétion, demande de chercher tous les items inactifs potentiellement utilisables $[B \rightarrow \gamma\bullet, j, k]$, introduisant un troisième indice libre k entre 1 et n. On obtient ainsi la complexité annoncée. Dans la pratique, les analyseurs obtenus sont "souvent" quasi-linéaires[6]. On peut finalement montrer que la complexité est dans le pire cas $O(n^2)$ pour des grammaires non-ambiguës.

Préfixe viable Une propriété remarquable de l'algorithme d'Earley est sa capacité à localiser sans délai les erreurs. Cette propriété, dite du *préfixe viable*, garantit qu'une erreur est détectée par l'algorithme lors du scan du premier symbole u_i tel que $u_1 \dots u_i$ n'est pas un préfixe d'au moins un mot de $L(G)$.

Preuve. C'est vrai pour les items initiaux. Supposons que ce soit vrai après insertion du nième item et considérons l'insertion d'un item supplémentaire. Deux cas sont à considérer :
– soit cet item provient d'une application de *pred* ou de *comp* : dans ce cas, le nouvel item $[B \rightarrow \bullet\alpha, i, j]$ ne couvre aucun symbole terminal supplémentaire ; la propriété du préfixe viable reste donc satisfaite.
– soit cet item provient d'une application de *scan* : ceci signifie que le symbole lu sur l'entrée courante avait été prédit et est donc susceptible de donner lieu à un développement de l'analyse.

12.3 Compléments

12.3.1 Vers le parsage déductif

Nous avons pour l'instant développé de manière relativement informelle les algorithmes de parsage tabulaire. Il nous reste à aborder maintenant une dernière question importante, qui est celle de la correction de ces algorithmes. Comment s'assurer, que formalisés sous la forme d'une technique de remplissage d'une table, ces algorithmes ne "manquent" pas des analyses ?

Informellement, deux conditions doivent être simultanément remplies pour assurer la validité de cette démarche :

[6] En particulier si la grammaire est déterministe !

- chaque analyse possible doit être représentable sous la forme d'un item de la table. C'est évidemment le cas avec les items que nous avons utilisé à la section précédente, qui expriment une analyse complète de $u_1 \dots u_n$ par un item de la forme $[S \rightarrow \alpha \bullet, 1, n]$.
- le mécanisme de déduction des items doit être tel que :
  - seuls sont déduits des items conformes à la stratégie de parsage ;
  - tous les items conformes sont déduits à une étape de donnée de l'algorithme : propriété de *complétude*.

  La notion de conformité introduite ici est relative aux propriétés qu'expriment les items. Ainsi, il est possible de montrer que tout item de l'algorithme d'Earley vérifie la propriété suivante (on parle également *d'invariant*) :

$$[A \rightarrow \alpha \bullet \beta, i, j] \in T \leftrightarrow \begin{cases} S \overset{\star}{\Rightarrow} & u_1 \dots u_{i-1} A \gamma \\ \alpha \rightarrow u_i \dots u_{j-1} \end{cases}$$

Lorsque ces propriétés sont remplies, on dit que l'algorithme est correct. Des résultats généraux concernant la correction des algorithmes tabulaires sont donnés dans (Sikkel and Nijholt, 1997), ou encore dans (Shieber et al., 1994), qui ré-exprime les stratégies tabulaires dans le cadre de systèmes de déductions logiques.

12.3.2 D'autres stratégies de parsage

L'alternative principale aux techniques tabulaires pour le parsage de grammaires ambiguës est le parsage LR généralisé (GLR) (Tomita, 1986). L'idée de base de cette approche conduire une analyse LR classique en gérant les ambiguïtés "en largeur d'abord". Rencontrant une case qui contient une ambiguïté, on duplique simplement la pile d'analyse pour développer en parallèle les multiples branches. S'en tenir là serait catastrophique, pouvant donner lieu à un nombre exponentiel de branches à maintenir en parallèle. Cette idée est "sauvée" par la factorisation des piles dans un graphe, permettant d'éviter de dupliquer les développements des piles qui partagent un même futur. Cette factorisation permet de récupérer la polynomialité de l'algorithme, qui, comme celui d'Earley a une complexité en $O(n^3)$ dans le pire cas, et souvent meilleure pour des grammaires « pas trop ambiguës ».

Les meilleurs résultats théoriques pour le parsage des grammaires CF sont donnés par l'algorithme proposé par dans (Valiant, 1975), qui promet une complexité en $O(2^{2.81})$, en ramenant le problème du parsage à celui du produit de matrices booléennes, pour lequel il existe des algorithmes extrêmement efficaces. Ce résultat n'est toutefois intéressant que d'un point de vue théorique, car la constante exhibée dans la preuve de Valiant est trop grande pour que le gain soit réellement significatif dans les applications les plus courantes.

Pour conclure avec les questions de la complexité, il est intéressant (et troublant) de noter que pour l'immense majorité des grammaires, on sait construire (parfois dans la douleur) des analyseurs linéaires. Par contre, on ne dispose pas aujourd'hui de méthode générale permettant de construire un analyseur linéaire pour n'importe quelle grammaire. En existe-t-il une ? La question est ouverte. Par contre, dans tous les cas, pour lister toutes les analyses, la complexité est exponentielle[7], tout simplement parce que le nombre d'analyses est dans le pire des cas exponentiel.

[7] Si l'on n'y prend garde les choses peuvent même être pire, avec des grammaires qui seraient infiniment ambiguës.