$$
\begin{aligned}
Z &\rightarrow S\# \\
S &\rightarrow TS' \\
S' &\rightarrow +TS' \mid \varepsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \varepsilon \\
F &\rightarrow (S) \mid D \\
D &\rightarrow 1 \mid 2
\end{aligned}
$$

TAB. 7.11 – Une grammaire LL pour les expressions arithmétiques

	FIRST						*FOLLOW*		
It.	1	2	3	4	5	It.	0	1	2
Z				(	(12	Z			
S			(	(12	(12	S	#)	#)	#)
S'	+	+	+	+	+	S'		#)	#)
T		(	(12	(12	(12	F	+	+#)	+#)
T'	*	*	*	*	*	F'		+#)	+#)
F	(	(12	(12	(12	(12	T	*	*+#)	*+#)
D	12	12	12	12	12	D		*+#)	*+#)

TAB. 7.12 – Calcul de *FIRST* et *FOLLOW*

Il est alors possible de déterminer la table d'analyse prédictive, représentée dans la Table 7.13. Cette table est sans conflit, puisque chaque case contient au plus une production.

	#	+	*	(	)	1	2
Z				$Z \rightarrow S\#$		$Z \rightarrow S\#$	$Z \rightarrow S\#$
S				$S \rightarrow FS'$		$S \rightarrow FS'$	$S \rightarrow FS'$
S'	$S' \rightarrow \varepsilon$	$S' \rightarrow +FS'$			$S' \rightarrow \varepsilon$		
T				$T \rightarrow TF'$		$T \rightarrow TF'$	$T \rightarrow TF'$
T'	$F' \rightarrow \varepsilon$	$F' \rightarrow \varepsilon$	$F' \rightarrow *F'T$	$F' \rightarrow \varepsilon$			
F				$T \rightarrow (S)$		$T \rightarrow D$	$T \rightarrow D$
D					$D \rightarrow 1$	$D \rightarrow 2$	

TAB. 7.13 – Table d'analyse prédictive pour la grammaire de l'arithmétique

Cette table permet effectivement d'analyser déterministiquement des expressions arithmétiques simples, comme le montre la trace d'exécution de la Table 7.14.

7.2 Analyseurs LR

7.2.1 Concepts

Comme évoqué à la section 6.2, les analyseurs ascendants cherchent à récrire le mot à analyser afin de se ramener, par des réductions successives, à l'axiome de la grammaire. Les bifurcations dans le graphe de recherche correspondent alors aux alternatives suivantes :

(i) la partie droite α d'une production $A \rightarrow \alpha$ est trouvée dans le proto-mot courant ; on choisit de remplacer α par A pour construire un nouveau proto-mot et donc d'appliquer une *réduction*.

Pile	Symbole	(...) Pile	Symbole
Z	2	2*(1T'S')T'S'#	+
S#	2	2*(1S')T'S'#	+
TS'#	2	2*(1+TS')T'S'#	+
FT'S'#	2	2*(1+FT'S')T'S'#	+
DT'S'#	2	2*(1+DT'S')T'S'#	+
2T'S'#	*	2*(1+2T'S')T'S'#	)
2*FT'S'#	(	2*(1+2S')T'S'#	)
2*(S)T'S'#	1	2*(1+2)T'S'#	#
2*(TS')T'S'#	1	2*(1+2)S'#	#
2*(FT'S')T'S'#	1	2*(1+2)#	#
2*(DT'S')T'S'#	1	succès	

TAB. 7.14 – Trace de l'analyse déterministe de $2 * (1 + 2)$

(ii) on poursuit l'examen du proto-mot courant en considérant un autre facteur α', obtenu, par exemple, en étendant α par la droite. Cette action correspond à un décalage de l'entrée.

Afin de rationaliser l'exploration du graphe de recherche correspondant à la mise en œuvre de cette démarche, commençons par décider d'une stratégie d'examen du proto-mot courant : à l'instar de ce que nous avons mis en œuvre dans l'algorithme 2, nous l'examinerons toujours depuis la gauche vers la droite. Si l'on note α le proto-mot courant, factorisé en $\alpha = \beta\gamma$, où β est la partie déjà examinée, l'alternative précédente se récrit selon :

– β possède un suffixe δ correspondant à la partie droite d'une règle : réduction et développement d'un nouveau proto-mot.
– β est étendu par la droite par décalage d'un nouveau terminal.

Cette stratégie conduit à la construction de dérivations *droites* de l'entrée courante. Pour vous en convaincre, remarquez que le suffixe γ du proto-mot courant n'est jamais modifié : il n'y a toujours que des terminaux à droite du symbole récrit, ce qui est conforme à la définition d'une dérivation droite.

Illustrons ce fait en considérant la grammaire de la Table 7.15.

$$
\begin{aligned}
S &\to AB \\
A &\to aA \mid b \\
B &\to bB \mid a
\end{aligned}
$$

TAB. 7.15 – Une grammaire pour $a^\star bb^\star a$

Soit alors $u = abba$; une analyse ascendante de ce mot est reproduite à la Table 7.16.

En considérant les réductions opérées, on obtient la dérivation : $S \Rightarrow_G AB \Rightarrow_G AbB \Rightarrow_G Aba \Rightarrow_G aAba \Rightarrow_G abba$, qui est effectivement une dérivation droite. On note également qu'on a introduit un troisième type d'action de l'analyseur, consistant à *accepter* l'entrée comme un mot de la grammaire. Il existe enfin un quatrième type d'action, non représenté ici, consistant à diagnostiquer une situation d'échec de l'analyse.

Dernière remarque concernant cette trace : les différentes transformations possibles de α via les actions de réduction et de décalage n'agissent que sur les suffixes de β. Ceci suggère d'implanter β sous la forme d'une pile, sur laquelle s'accumulent (puis se réduisent) progressivement les symboles de u. Si l'on adopte ce point de vue, une pile correspondant à la trace de la Table 7.16

α	β	γ	Action
$abba$	ε	$abba$	décaler
$abba$	a	bba	décaler
$abba$	ab	ba	réduire par $A \to b$
$aAba$	aA	ba	réduire par $A \to aA$
Aba	A	ba	décaler
Aba	Ab	a	décaler
Aba	Aba	ε	réduire par $B \to a$
AbB	AbB	ε	réduire par $B \to bB$
AB	AB	ε	réduire par $S \to AB$
S			fin : accepter l'entrée

TAB. 7.16 – Analyse ascendante de $u = abba$

passerait par les états successifs suivants : $a, ab, aA, A, Ab, Aba, AbB, AB, S$. Bien naturellement, cette implantation demanderait également de conserver un pointeur vers la position courante dans u, afin de savoir quel symbole empiler. Dans la suite, nous ferons l'hypothèse que β est effectivement implanté sous la forme d'une pile.

Comment procéder pour rendre ce processus déterministe ? Cherchons des éléments de réponse dans la trace d'analyse présentée dans la Table 7.16. Ce processus contient quelques actions déterministes, comme la première opération de décalage : lorsqu'en effet β (la pile) ne contient aucun suffixe correspondant à une partie droite de production, décaler est l'unique action possible.

De manière duale, lorsque γ est vide, décaler est impossible : il faut nécessairement réduire, lorsque cela est encore possible : c'est, par exemple, ce qui est fait durant la dernière réduction.

Un premier type de choix correspond au second décalage : $\beta = a$ est à ce moment égal à une partie droite de production ($B \to a$), pourtant on choisit ici de décaler (ce choix est presque toujours possible) plutôt que de réduire. Notons que la décision prise ici est la (seule) bonne décision : réduire prématurément aurait conduit à positionner un B en tête de β, conduisant l'analyse dans une impasse : B n'apparaissant en tête d'aucune partie droite, il aurait été impossible de le réduire ultérieurement. Un second type de configuration (non représentée ici) est susceptible de produire du non-déterminisme : il correspond au cas où l'on trouve en queue de β deux parties droites de productions : dans ce cas, il faudra choisir entre deux réductions concurrentes.

Résumons-nous : nous voudrions, de proche en proche, et par simple consultation du sommet de la pile, être en mesure de décider déterministiquement si l'action à effectuer est un décalage ou bien une réduction (et dans ce cas quelle est la production à utiliser), ou bien encore si l'on se trouve dans une situation d'erreur. Une condition suffisante serait que, pour chaque production p, on puisse décrire l'ensemble L_p des configurations de la pile pour lesquelles une réduction par p est requise ; et que, pour deux productions p_1 et p_2 telles que $p_1 \neq p_2$, L_{p_1} et L_{p_2} soient toujours disjoints. Voyons à quelle(s) condition(s) cela est possible.

7.2.2 Analyseurs LR(0)

Pour débuter, formalisons la notion de contexte LR(0) d'une production.

Définition 7.6. *Soit G une grammaire hors-contexte, et $p = (A \to \alpha)$ une production de G, on appelle*

contexte LR(0) de p le langage $L_{A\to\alpha}$ défini par :

$$L_{A\to\alpha} = \{\gamma = \beta\alpha \in (\Sigma \cup V)^\star \ tq.\ \exists v \in \Sigma^\star, S \overset{\star}{\Rightarrow}_G \ \beta A v \Rightarrow_G \ \beta\alpha v\}$$

En d'autres termes, tout mot γ de $L_{A\to\alpha}$ contient un suffixe α et est tel qu'il existe une réduction de γv en S qui débute par la réduction de α en A. Chaque fois qu'un tel mot γ apparaît dans la pile d'un analyseur ascendant gauche-droit, il est utile d'opérer la réduction $A \to \alpha$; à l'inverse, si l'on trouve dans la pile un mot absent de $L_{A\to\alpha}$, alors cette réduction ne doit pas être considérée, même si ce mot se termine par α.

Examinons maintenant de nouveau la grammaire de la Table 7.15 et essayons de calculer les langages L_p pour chacune des productions. Le cas de la première production est clair : il faut impérativement réduire lorsque la pile contient AB, et c'est là le seul cas possible. On déduit directement que $L_{S\to AB} = \{AB\}$. Considérons maintenant $A \to aA$: la réduction peut survenir quel que soit le nombre de a présents dans la pile. En revanche, si la pile contient un symbole différent de a, c'est qu'une erreur aura été commise. En effet :
– une pile contenant une séquence baA ne pourra que se réduire en AA, dont on ne sait plus que faire ; ceci proscrit également les piles contenant plus d'un A, qui aboutissent pareillement à des configurations d'échec ;
– une pile contenant une séquence $B \ldots A$ ne pourra que se réduire en BA, dont on ne sait non plus comment le transformer.
En conséquence, on a : $L_{A\to aA} = aa^\star A$. Des considérations similaires nous amènent à conclure que :
– $L_{A\to b} = a^\star b$;
– $L_{B\to bB} = Abb^\star B$;
– $L_{B\to a} = Ab^\star a$;
Chacun des ensembles L_p se décrivant par une expression rationnelle, on déduit que l'ensemble des L_p se représente sur l'automate de la Figure 7.1, qui réalise l'union des langages L_p. Vous noterez de plus que (i) l'alphabet d'entrée de cet automate contient à la fois des symboles terminaux et non-terminaux de la grammaire ; (ii) les états finaux de l'automate de la Figure 7.1 sont associés aux productions correspondantes ; (iii) toute situation d'échec dans l'automate correspond à un échec de l'analyse : en effet, ces configurations sont celles où la pile contient un mot dont l'extension ne peut aboutir à aucune réduction : il est alors vain de continuer l'analyse.

Comment utiliser cet automate ? Une approche naïve consiste à mettre en œuvre la procédure suivante : partant de l'état initial $q_i = q_0$ et d'une pile vide on applique des décalages jusqu'à atteindre un état final q. On réduit ensuite la pile selon la production associée à q, donnant lieu à une nouvelle pile β qui induit un repositionnement dans l'état $q_i = \delta^\star(q_0, \beta)$ de l'automate. La procédure est itérée jusqu'à épuisement simultané de la pile et de u. Cette procédure est formalisée à travers l'algorithme 7.

L'implantation décrite dans l'algorithme 7 est un peu naïve : en effet, à chaque réduction, on se repositionne à l'état initial de l'automate, perdant ainsi le bénéfice de l'analyse des symboles en tête de la pile. Une implantation plus efficace consiste à mémoriser, pour chaque symbole de la pile, l'état q atteint dans A. A la suite d'une réduction, on peut alors directement se positionner sur q pour poursuivre l'analyse. Cette amélioration est mise en œuvre dans l'algorithme 8.

Notez, pour finir, que l'on effectue en fait deux sortes de transitions dans A : celles qui sont effectuées directement à l'issue d'un décalage et qui impliquent une extension de la pile ; et celles qui sont effectuées à l'issue d'une réduction et qui consistent simplement à un repositionnement ne nécessitant pas de nouveau décalage. Ces deux actions sont parfois distinguées, l'une sous le nom de décalage (shift), l'autre sous le nom de *goto*.

De l'automate précédent, se déduit mécaniquement une table d'analyse (dite LR(0)), donnée pour

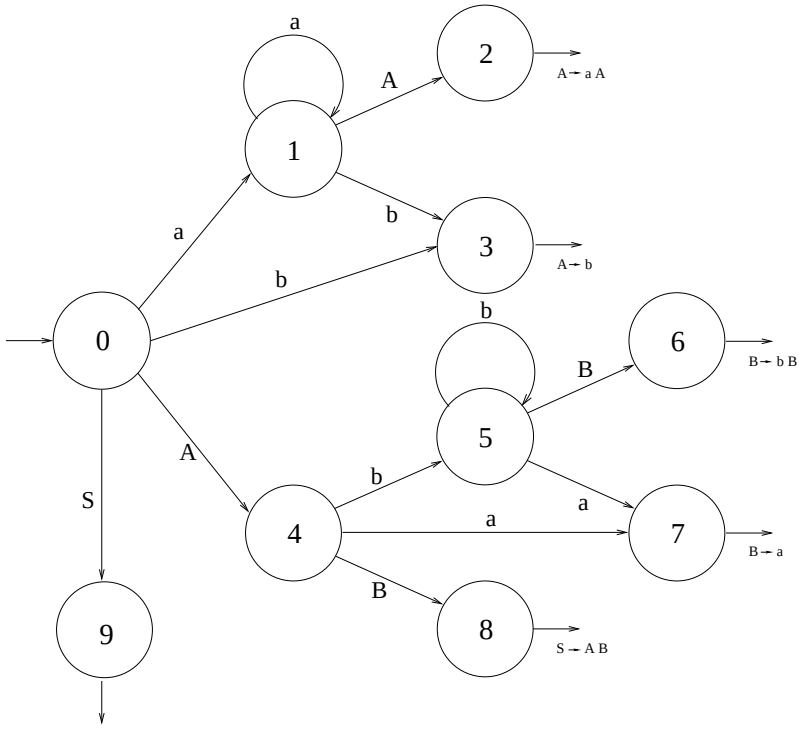

FIG. 7.1 – L'automate des réductions licites

Algorithm 7 – Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup V, Q, q_0, F, \delta)$. Version 1

// initialisation
$\beta := \varepsilon$ // Initialisation de la pile
$q := q_0$ // Se positionner dans l'état initial
$i := j := 0$
while $(i \leq |u|)$ **do**
    while $(j \leq |\beta|)$ **do**
        $j := j + 1$
        $q := \delta(q, \beta_j)$
    od
    while $(q \notin F)$ **do**
        $\beta := \beta u_i$ // Empilage de u_i
        if ($\delta(q, u_i)$ existe) **then** $q := \delta(q, u_i)$ **else** *return*(**false**) **fi**
        $i := i + 1$
    od
    // Réduction de $p : A \rightarrow \gamma$
    $\beta := \beta\gamma^{-1}A$
    $j := 0$
    $q := q_0$
od
if $(\beta = S \ \wedge \ q \in F)$ **then** *return*(**true**) **else** *return*(**false**) **fi**

Algorithm 8 – Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup V, Q, q_0, F, \delta)$. Version 2

// initialisations
$\beta := (\varepsilon, q_0)$ // Initialisation de la pile
$q := q_0$ // Se positionner dans l'état initial
$i := 0$
while ($i \leq |u|$) **do**
    while ($q \notin F$) **do**
        if ($\delta(q, u_i)$ existe) **then do**
            $q := \delta(q, u_i)$ // Progression dans A
            $push(\beta, (u_i, q))$ // Empilage de (u_i, q)
            $i := i + 1$
        od
        else *return*(**false**)
        fi
    od
    // On atteint un état final : réduction de $p : A \rightarrow \delta$
    $j := 0$
    // Dépilage de δ
    while ($j < |\delta|$) **do**
        $pop(\beta)$
        $j := j + 1$
    od
    // (x, q) est sur le sommet de la pile : repositionnement
    $push(\beta, (A, \delta(q, A)))$
    $q := \delta(q, A)$
od
if ($\beta = (S, q) \ \wedge \ q \in F$) **then** *return*(**true**) **else** *return*(**false**) **fi**

l'automate de la Figure 7.1 à la Table 7.17. Cette table résume les différentes actions à effectuer en fonction de l'état courant de l'automate.

La Table 7.17 contient trois types d'entrées :
– une entrée de type (s, i) en colonne x signifie : effectuer un décalage, lire le symbole en tête de pile, si c'est un x transiter dans l'état i. *Attention* : le décalage a bien lieu de *manière inconditionnelle*, c'est-à-dire indépendamment de la valeur du symbole empilé ; en revanche, cette valeur détermine la transition de l'automate à exercer ;
– une entrée de type (g, j) en colonne X signifie : lire le symbole en tête de pile, si c'est un X transiter dans l'état j.
– une entrée de type $(r, A \rightarrow \alpha)$ en colonne $*$ signifie : réduire la pile selon $A \rightarrow \alpha$.
Toutes les autres configurations (qui correspondent aux cases vides de la table) sont des situations d'erreur. Pour distinguer, dans la table, les situations où l'analyse se termine avec succès, il est courant d'utiliser la transformation suivante :
– on ajoute un nouveau symbole terminal, par exemple # ;
– on transforme G en G' en ajoutant un nouvel axiome Z et une nouvelle production $Z \rightarrow S\#$, où S est l'axiome de G.
– on transforme l'entrée à analyser en $u\#$;

	A	B	a	b	*
0	g,4		s,1	s,3	
1	g,2		s,1	s,3	
2					r, $A \rightarrow aA$
3					r, $A \rightarrow b$
4		g,8	s,7	s,5	
5		g,6	s,7	s,5	
6					r, $B \rightarrow bB$
7					r, $B \rightarrow a$
8					r, $S \rightarrow AB$
	A	B	a	b	

TAB. 7.17 – Une table d'analyse LR(0)

– l'état (final) correspondant à la réduction $Z \rightarrow S\#$ est (le seul) état d'acceptation, puisqu'il correspond à la fois à la fin de l'examen de u (# est empilé) et à la présence du symbole S en tête de la pile.

Il apparaît finalement, qu'à l'aide de la Table 7.17, on saura analyser la grammaire de la Table 7.15 de manière déterministe et donc avec une complexité linéaire. Vous pouvez vérifier cette affirmation en étudiant le fonctionnement de l'analyseur pour les entrées $u = ba$ (succès), $u = ab$ (échec), $u = abba$ (succès).

Deux questions se posent alors : (i) peut-on, pour toute grammaire, construire un tel automate ? (ii) comment construire l'automate à partir de la grammaire G ? La réponse à (i) est non : il existe des grammaires (en particulier les grammaires ambiguës) qui résistent à toute analyse déterministe. Il est toutefois possible de chercher à se rapprocher de cette situation, comme nous le verrons à la section 7.2.3. Dans l'intervalle, il est instructif de réfléchir à la manière de construire automatiquement l'automate d'analyse A.

L'intuition de la construction se fonde sur les remarques suivantes. Initialement, on dispose de u, non encore analysé, qu'on aimerait pouvoir réduire en S, par le biais d'une série non-encore déterminée de réductions, mais qui s'achèvera nécessairement par une réduction de type $S \rightarrow \alpha$.

Supposons, pour l'instant, qu'il n'existe qu'une seule S-production : $S \rightarrow X_1 \dots X_k$: le but original de l'analyse "aboutir à une pile dont S est l'unique symbole en ayant décalé tous les symboles de u" se reformule alors en : "aboutir à une pile égale à $X_1 \dots X_k$ en ayant décalé tous les symboles de u". Ce nouveau but se décompose naturellement en une série d'étapes qui vont devoir être accomplies séquentiellement : d'abord parvenir à une configuration où X_1 est «au fond» de la pile, puis faire que X_2 soit empilé juste au-dessus de X_1...

Conserver la trace de cette série de buts suggère d'insérer dans l'automate d'analyse une branche correspondant à la production $S \rightarrow X_1 \dots X_k$, qui s'achèvera donc sur un état final correspondant à la réduction de $X_1 \dots X_k$ en S. Une telle branche est représentée à la Figure 7.2.

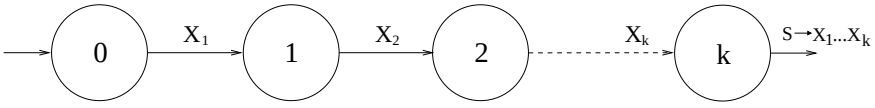

FIG. 7.2 – Une branche de l'automate

Chaque état le long de cette branche correspond à la résolution d'un sous-but supplémentaire, la transition X_i annonçant l'empilage de X_i, et déclenchant la recherche d'un moyen d'empiler X_{i+1}.

Pour formaliser cette idée, nous introduisons le concept de *production (ou règle) pointée* [3].

Définition 7.7. *Une production pointée d'une grammaire hors-contexte G est un triplet (A, α, β) de $V \times (V \cup \Sigma)^\star \times (V \cup \Sigma)^\star$, avec $A \to \gamma = \alpha\beta$ une production de G. Une production pointée est notée avec un point : $A \to \alpha \bullet \beta$.*

Une production pointée exprime la résolution partielle d'un but : $A \to \alpha \bullet \beta$ exprime que la résolution du but "empiler A en terminant par la réduction $A \to \alpha\beta$" a été partiellement accomplie, en particulier que α a déjà été empilé et qu'il reste encore à empiler les symboles de β. Chaque état de la branche de l'automate correspondant à la production $A \to \gamma$ s'identifie ainsi à une production pointée particulière, les états initiaux et finaux de cette branche correspondant respectivement à : $A \to \bullet\gamma$ et $A \to \gamma\bullet$.

Retournons à notre problème original, et considérons maintenant le premier des sous-buts : "empiler X_1 au fond de la pile". Deux cas de figure sont possibles :
- soit X_1 est symbole terminal : le seul moyen de l'empiler consiste à effectuer une opération de décalage, en cherchant un tel symbole en tête de la partie non encore analysée de l'entrée courante.
- soit X_1 est un non-terminal : son insertion dans la pile résulte nécessairement d'une série de réductions, dont la dernière étape concerne une X_1-production : $X_1 \to Y_1 \dots Y_n$. De nouveau, le but "observer X_1 au fond de la pile" se décompose en une série de sous-buts, donnant naissance à une nouvelle «branche» de l'automate pour le mot $Y_1 \dots Y_n$. Comment relier ces deux branches ? Tout simplement par une transition εentre les deux états initiaux, indiquant que l'empilage de X_1 se résoudra en commençant l'empilage de Y_1 (voir la Figure 7.3). S'il existe plusieurs X_1-productions, on aura une branche (et une transition ε) par production.

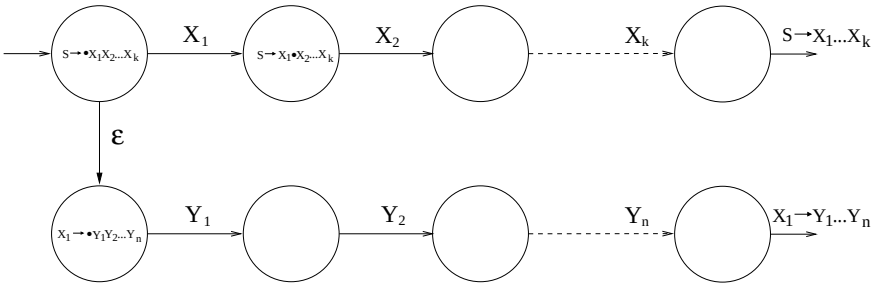

FIG. 7.3 – Deux branches de l'automate

Ce procédé se généralise : chaque fois qu'une branche porte une transition $\delta(q, X) = r$, avec X un non-terminal, il faudra ajouter une transition entre q et tous les états initiaux des branches correspondants aux X-productions.

De ces remarques, découle un procédé systématique pour construire un ε-NFA $(V \cup \Sigma, Q, q_0, F, \delta)$ permettant de guider une analyse ascendante à partir d'une grammaire $G = (\Sigma, V, Z, P)$, telle que Z est non-récursif et ne figure en partie gauche que dans l'unique règle $Z \to \alpha$:
- $Q = \{[A \to \alpha \bullet \beta] \text{ avec } A \to \alpha\beta \in P\}$
- $q_0 = [Z \to \bullet\alpha]$
- $F = \{[A \to \alpha\bullet] \text{ avec } A \to \alpha \in P\}$; à chaque état final est associée une production $A \to \alpha$
- $\forall q = [A \to \alpha \bullet X\beta] \in Q, \delta(q, X) = [A \to \alpha X \bullet \beta]$
- $\forall q = [A \to \alpha \bullet B\beta] \in Q$ tq. $B \in V, \forall q' = [B \to \bullet\gamma], \delta(q, \varepsilon) = q'$.

Le théorème suivant, non démontré ici, garantit que ce procédé de construction permet effectivement d'identifier les contextes LR(0) des productions.

[3] On trouve également le terme *d'item* et en anglais de *dotted rule*.

Théorème 7.2. *Soit A l'automate fini dérivé de G par la construction précédente, alors : $\delta^\star(q_0, \gamma) = [A \to \alpha \bullet \beta]$ si et seulement si :*
– *(i)* $\exists \delta, \gamma = \delta\alpha$
– *(ii)* $\gamma\beta \in L_{A \to \alpha\beta}$

Ce résultat assure, en particulier, que lorsque l'on atteint un état final de l'automate (pour $\beta = \varepsilon$), la pile contient effectivement un mot appartenant au contexte LR(0) de la production associée à l'état final atteint.

Il est naturellement possible de déterminiser cet automate, en appliquant les algorithmes de suppression des transitions spontanées (voir page 33), puis la construction des sous-ensembles décrite à la page 30 ; on prendra soin de propager lors de ces transformations l'information de réduction associée aux états finaux des branches. En clair, si un état q du déterminisé contient une production pointée originale de type $A \to \alpha\bullet$, alors q sera un état (final) dans lequel cette réduction est possible. Le lecteur est vivement encouragé à entreprendre cette démarche et vérifier qu'il retrouve bien, en partant de la grammaire de la Table 7.15 un automate ressemblant fortement à celui de la Figure 7.1. On déduit finalement de cet automate déterministe, par le procédé utilisé pour construire la Table 7.17, la *table d'analyse LR(0)*.

Définition 7.8. *Une grammaire hors-contexte est LR(0)*[4] *si sa table $T()$ d'analyse LR(0) est telle que : pour toute ligne i (correspondant à un état de l'automate), soit il existe $x \in V \cup \Sigma$ tel que $T(i, x)$ est non-vide et $T(i, *)$ est vide ; soit $T(i, *)$ est non-vide et contient une réduction unique.*

Une grammaire LR(0) peut être analysée de manière déterministe. La définition d'une grammaire LR(0) correspond à des contraintes qui sont souvent, dans la pratique, trop fortes pour des grammaires «réelles», pour lesquelles la construction de la table LR(0) aboutit à des conflits. Le premier type de configuration problématique correspond à une indétermination entre décaler et réduire (conflit *shift/reduce*) ; le second type correspond à une indétermination sur la réduction à appliquer ; on parle alors de conflit *reduce/reduce*. Vous noterez, en revanche, qu'il n'y a jamais de conflit *shift/shift*. Pourquoi ?

Comme pour les analyseurs descendants, il est possible de lever certaines indéterminations en s'autorisant un regard en avant sur l'entrée courante ; les actions de l'analyseur seront alors conditionnées non seulement par l'état courant de l'automate d'analyse, mais également par les symboles non-analysés. Ces analyseurs font l'objet de la section qui suit.

7.2.3 Analyseurs LR(1), LR(k)...

Regard avant

Commençons par étudier le cas le plus simple, celui où les conflits peuvent être résolus avec un regard avant de 1. C'est, par exemple, le cas de la grammaire de la Table 7.18 :

Considérons une entrée telle que $x+x+x$: sans avoir besoin de construire la table LR(0), il apparaît qu'après avoir empilé le premier x, puis l'avoir réduit en T, par $T \to x$, deux alternatives vont s'offrir à l'analyseur :
– soit immédiatement réduire T en E
– soit opérer un décalage et continuer de préparer une réduction par $E \to T + E$.

[4] Un 'L' pour *left-to-right*, un 'R' pour *rightmost derivation*, un 0 pour *0 lookahead* ; en effet, les décisions (décalage vs. réduction) sont toujours prises étant uniquement donné l'état courant (le sommet de la pile).

$$S \rightarrow E\#$$
$$E \rightarrow T + E \mid T$$
$$T \rightarrow x$$

TAB. 7.18 – Une grammaire non-LR(0)

Il apparaît pourtant que '+' ne peut jamais suivre E dans une dérivation réussie : vous pourrez le vérifier en construisant des dérivations droites de cette grammaire. Cette observation anticipée du prochain symbole à empiler suffit, dans le cas présent, à restaurer le déterminisme. Comment traduire cette intuition dans notre analyseur ?

La réponse consiste à modifier la procédure de construction de l'automate décrite à la section précédente en choisissant comme ensemble d'états toutes les paires[5] constituées d'une production pointée et d'un terminal. On note ces états $[A \rightarrow \beta \bullet \gamma, a]$. La construction de l'automate d'analyse LR (ici LR(1)) $(V \cup \Sigma, Q, q_0, F, \delta)$ se déroule alors comme suit (on suppose toujours que l'axiome Z n'est pas récursif et n'apparaît que dans une seule règle) :

- $Q = \{[A \rightarrow \alpha \bullet \beta, a]$ avec $A \rightarrow \alpha\beta \in P$ et $a \in \Sigma\}$
- $q_0 = [Z \rightarrow \bullet\alpha, ?]$, où ? est un symbole «joker» qui vaut pour n'importe quel symbole terminal ;
- $F = \{[A \rightarrow \alpha\bullet, a]$, avec $A \rightarrow \alpha \in P\}$; à chaque état final est associée la production $A \rightarrow \alpha$, correspondant à la réduction à opérer ;
- $\forall q = [A \rightarrow \alpha \bullet X\beta, a] \in Q, \delta(q, X) = [A \rightarrow \alpha X \bullet \beta, a]$; comme précédemment, ces transitions signifient la progression de la résolution du but courant par empilement de X ;
- $\forall q = [A \rightarrow \alpha \bullet B\beta, a] \in Q$ tq. $B \in V, \forall q' = [B \rightarrow \bullet\gamma, b]$ tq. $b \in FIRST(\beta a), \delta(q, \varepsilon) = q'$.
  La condition supplémentaire $b \in FIRST(\beta a)$ (voir la section 7.1.3) introduit une information de désambiguïsation qui permet de mieux caractériser les réductions à opérer. En particulier, dans le cas LR(1), on espère qu'en prenant en compte la valeur du premier symbole dérivable depuis βa , il sera possible de sélectionner la bonne action à effectuer en cas de conflit sur B.
  Pour illustrer ce point, considérons de nouveau la grammaire précédente 7.18. L'état initial de l'automate LR(1) correspondant est $[S \rightarrow \bullet E\#]$; lequel a deux transitions ε, correspondant aux deux façons d'empiler un E, atteignant respectivement les états $q_1 = [E \rightarrow \bullet T + E, \#]$ et $q_2 = [E \rightarrow \bullet T, \#]$. Ces deux états se distinguent par leurs transitions εsortantes : dans le cas de q_1, vers $[T \rightarrow \bullet\, x, +]$; dans le cas de q_2, vers $[T \rightarrow \bullet\, x, \#]$. À l'issue de la réduction d'un x en T, le regard avant permet de décider sans ambiguïté de l'action : si c'est un '+', il faut continuer de chercher une réduction $E \rightarrow T + E$; si c'est un '#', il faut réduire selon $E \rightarrow T$.

Construction pas-à-pas de l'automate d'analyse

La construction de la table d'analyse à partir de l'automate se déroule comme pour la table LR(0). Détaillons-en les principales étapes : après construction et déterminisation de l'automate LR(1), on obtient un automate fini dont les états finaux sont associés à des productions de G. On procède alors comme suit :

- pour chaque transition de q vers r étiquetée par un terminal a, la case $T(q, a)$ contient la séquence d'actions (décaler, consommer a en tête de la pile, aller en r) ;
- pour chaque transition de q vers r étiquetée par un non-terminal A, la case $T(q, A)$ contient la séquence d'actions (consommer A en tête de la pile, aller en r) ;
- pour chaque état final $q = [A \rightarrow \alpha\bullet, a]$, la case $T(q, a)$ contient l'unique action (réduire la pile selon $A \rightarrow \alpha$) : la décision de réduction (ainsi que la production à appliquer) est maintenant

[5] En fait, les prendre *toutes* est un peu excessif, comme il apparaîtra bientôt.

conditionnée par la valeur du regard avant associé à q.
Lorsque $T()$ ne contient pas de conflit, la grammaire est dite LR(1). Il existe des grammaires LR(1) ou «presque[6]» LR(1) pour la majorité des langages informatiques utilisés dans la pratique.

En revanche, lorsque la table de l'analyseur LR(1) contient des conflits, il est de nouveau possible de chercher à augmenter le regard avant pour résoudre les conflits restants. Dans la pratique[7], toutefois, pour éviter la manipulation de tables trop volumineuses, on préférera chercher des moyens ad-hoc de résoudre les conflits dans les tables LR(1) plutôt que d'envisager de construire des tables LR(2) ou plus. Une manière courante de résoudre les conflits consiste à imposer des priorités via des règles du type : "en présence d'un conflit shift/reduce, toujours choisir de décaler[8]...

En guise d'application, le lecteur est invité à s'attaquer à la construction de la table LR(1) pour la Table 7.18 et d'en déduire un analyseur déterministe pour cette grammaire. Idem pour la grammaire de la Table 7.19, qui engendre des mots tels que $x = * * x$.

$$
\begin{aligned}
S' &\rightarrow S\# \\
S &\rightarrow V = E \mid E \\
E &\rightarrow V \\
V &\rightarrow *E \mid x
\end{aligned}
$$

Tab. 7.19 – Une grammaire pour les manipulations de pointeurs

Pour vous aider dans votre construction, la Figure 7.4 permet de visualiser l'effet de l'ajout du regard avant sur la construction de l'automate d'analyse.

Une construction directe

Construire manuellement la table d'analyse LR() est fastidieux, en particulier à cause du passage par un automate non-déterministe, qui implique d'utiliser successivement des procédures de suppression des transitions spontanées, puis de déterminisation. Dans la mesure où la forme de l'automate est très stéréotypée, il est possible d'envisager la construction directe de l'automate déterminisé à partir de la grammaire, en s'aidant des remarques suivantes :
– chaque état du déterminisé est un ensemble d'éléments de la forme [production pointée, terminal] : ceci du fait de l'application de la construction des sous-ensembles (cf. la section 3.1.4)
– la suppression des transitions spontanées induit la notion de *fermeture* : la ε-fermeture d'un état étant l'ensemble des états atteints par une ou plusieurs transitions ε.
Cette procédure de construction du DFA A est détaillée dans l'algorithme 9, qui utilise deux fonctions auxiliaires pour effectuer directement la déterminisation.

[6] C'est-à-dire que les tables correspondantes sont presque sans conflit

[7] Il existe une autre raison, théorique celle-là, qui justifie qu'on se limite aux grammaires LR(1) : les grammaires LR(1) engendrent tous les langages hors-contextes susceptibles d'être analysés par une procédure déterministe ! Nous aurons l'occasion de revenir en détail sur cette question au chapitre 9. En d'autres termes, l'augmentation du regard avant peut conduire à des grammaires plus simples à analyser ; mais ne change rien à l'expressivité des grammaires. La situation diffère donc ici de ce qu'on observe pour la famille des grammaires LL(k), qui induit une hiérarchie stricte de langages.

[8] Ce choix de privilégier le décalage sur la réduction n'est pas innocent : en cas d'imbrication de structures concurrentes, il permet de privilégier la structure la plus intérieure, ce qui correspond bien aux atteintes des humains. C'est ainsi que le mot `if cnd1 then if cond2 then inst1 else inst2` sera plus naturellement interprétée `if cnd1 then (if cond2 then inst1 else inst2)` que `if cnd1 then (if cond2 then inst1) else inst2` en laissant simplement la priorité au décalage du `else` qu'à la réduction de `if cond2 then inst1`. b

Algorithm 9 – Construction de l'automate d'analyse LR(1)

```
// Programme principal
begin
   q_0 = Closure([S' → •S#, ?]) // L'état initial de A
   Q := {q_0} // Les états de A
   T := ∅ // Les transitions de A
   while (true) do
         Q_i := Q
         T_i := T
         foreach q ∈ Q do  // q est lui-même un ensemble !
            foreach [A → β • Xγ, a] ∈ q do
               r := Successor(q, X)
               Q := Q ∪ {r}
               T := T ∪ {(q, X, r)}
            od
         od
         // test de stabilisation
         if (Q = Q_i ∧ T = T_i) then break fi
   od
end
// Procédures auxiliaires
Closure(q) // Construction directe de la ε-fermeture
begin
   while (true) do
         q_i := q
         foreach [A → β • Xγ, a] ∈ q do
            foreach (X → α) ∈ P do
               foreach b ∈ FIRST(γa) do
                  q := q ∪ [X → •α, b]
               od
            od
         od
         // test de stabilisation
         if (q = q_i) then break fi
   od
   return(q)
end
Successor(q, X) // Développement des branches
begin
   r := ∅
   foreach [A → β • Xγ, a] ∈ q do
      r := r ∪ [A → βX • γ, a]
   od
   return(Closure(r))
end
```

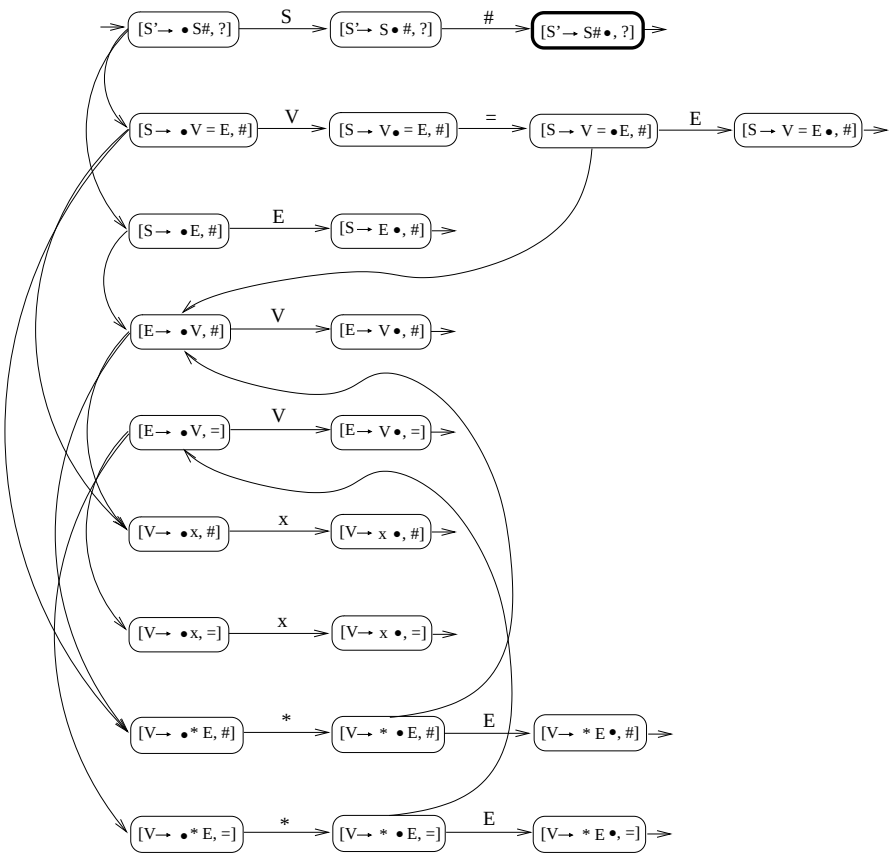

FIG. 7.4 – Les réductions licites pour la Table 7.19
Pour améliorer la lisibilité, les εne sont pas représentés. L'état d'acceptation est représenté en gras.

En guise d'application, vous vérifierez que la mise en œuvre de l'algorithme 9 construit directement le déterminisé de l'automate de la Figure 7.4.

7.2.4 Compléments

LR et LL La famille des analyseurs LR(k) permet d'analyser tous les langages LL(k) et bien d'autres langages non-ambigus. La raison de cette plus grande généricité des analyseurs LR est au fond leur plus grande «prudence» : alors qu'un analyseur LL(k) doit pouvoir sélectionner sans erreur une production $A \rightarrow \alpha$ sur la seule base des k symboles terminaux non encore appariés (dont tout ou partie peut être dérivé de A), un analyseur LR(k) fonde sa décision d'appliquer une réduction $A \rightarrow \alpha$ sur (i) la connaissance de l'intégralité de la partie droite α et (ii) la connaissance des k symboles terminaux à droite de A. Pour k fixé, il est alors normal qu'un analyseur LR(k), ayant plus d'information à sa disposition qu'un analyseur LL(k), fasse des choix plus éclairés, faisant ainsi porter moins de contraintes sur la forme de la grammaire.

Génération d'analyseurs Lorsque l'on s'intéresse à des grammaires réelles, la construction de l'automate et de la table d'analyse LR peut rapidement conduire à de très gros automates : il est en fait nécessaire de déterminiser un automate dont le nombre d'états est proportionnel à la somme des longueurs des parties droites des productions de G ; étape qui peut conduire (cf. la section 3.1.4) à un automate déterministe ayant exponentiellement plus d'états que le non-déterministe d'origine.

Il devient alors intéressant de recourir à des programmes capables de construire automatiquement un analyseur pour une grammaire LR : il en existe de nombreux, dont le plus fameux, `yacc` est disponible et documenté (dans sa version libre, connue sous le nom de `bison`) à l'adresse suivante : `http://www.gnu.org/software/bison/bison.html`.

LR et LALR et ... Utiliser un générateur d'analyseur tel que `bison` ne fait que reporter sur la machine la charge de construire (et manipuler) un gros automate ; ce qui, en dépit de l'indéniable bonne volonté générale des machines, peut malgré tout poser problème. Le remède le plus connu est d'essayer de compresser *avec perte*, lorsque cela est possible (et c'est le cas général) les tables d'analyse LR(1), donnant lieu à la classe d'analyseurs LALR(1), qui sont ceux que construit `yacc`. Il existe de nombreuses autres variantes des analyseurs LR visant à fournir des solutions pour sauver le déterminisme de l'analyse, tout en maintenant les tables dans des proportions raisonnables. Nous aurons l'occasion de revenir beaucoup plus en détail sur les analyseurs (*LA*)*LR* et leurs multiples variantes au chapitre 9.