1. *Soit la grammaire suivante : $S \to E\$ \; ; E \to bAc \mid Aa \mid bda \; ; A \to d$*

(a) *Construire les ensembles premier et suivant.*

	prem	suiv
S	b, d	-
E	b, d	\$
A	d	a, c, \$

(b) *La grammaire est-elle LL(1) ?*

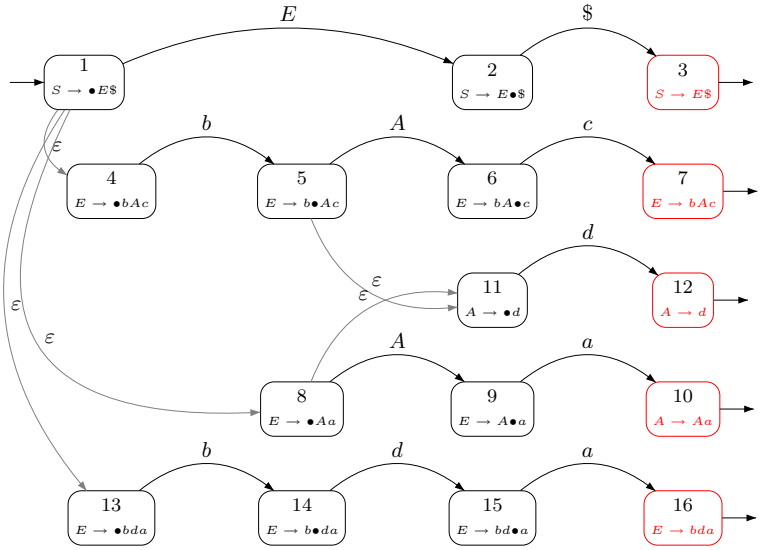
Pour répondre à la question on peut construire la table LL(1), ce qui est facile à partir des ensembles prem et suiv (d'autant plus facile qu'il n'y a pas d'ε-productions) :

	a	b	c	d	\$
S		E\$		E\$	
E		bAc bda		Aa	
A				d	

Il y a dans cette table une case comprenant deux entrées, cette grammaire n'est donc pas LL(1). On peut facilement se convaincre qu'elle n'est pas non plus LL(2), la même ambiguïté se retrouvant pour la chaîne bd. En revanche, elle est LL(3).

(c) *Construire l'automate des contextes LR(0). Cette grammaire est-elle LR(0) ?*

L'automate des contextes LR(0) peut être construit « à main levée », la grammaire étant relativement simple (mais attention à faire un automate déterministe), ou en appliquant l'algorithme qui établit une correspondance entre un état et une règle pointée. Dans ce cas, il est nécessaire de déterminiser l'automate résultant.



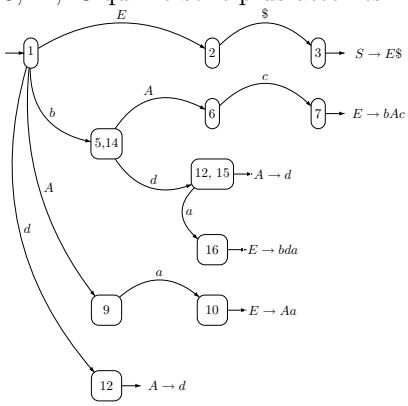
La déterminisation passe par la suppression des ε-transitions. Voir les tables :

	E	A	\$	a	b	c	d	$\varepsilon+$
→ 1	2		3					4, 8, 11, 13
2								
← 3								
4					5			
5		6						11
6						7		
← 7								
8		9						11
9				10				
← 10								
11							12	
← 12								
13					14			
14							15	
15				16				
16								

	E	A	\$	a	b	c	d
→ 1	2	**9**			**5,14**		**12**
2			3				
← 3							
4					5		
5		6					**12**
6						7	
← 7							
8		9					**12**
9				10			
← 10							
11							12
← 12							
13					14		
14							15
15				16			
16							

Il n'y a plus d'ε-transitions, mais il reste du non-déterminisme, qu'il faut réduire, ce qui, par la même occasion, nous débarrasse des états $4, 8, 11, 13$ qui ne sont plus atteints.

	E	A	\$	a	b	c	d
$\rightarrow 1$	2	9			$(5,14)$		12
2			3				
9				10			
$(5,14)$		6					$(12,15)$
$\leftarrow 12$							
$\leftarrow 3$							
$\leftarrow 10$							
4					5		
6						7	
$\leftarrow (12,15)$				16			
5		6					12
$\leftarrow 7$							
16							



L'automate LR(0) fait apparaître un conflit shift-reduce : à l'état (12,15) on peut soit réduire la pile (règle $A \rightarrow d$), soit faire un shift. La grammaire n'est donc pas LR(0).

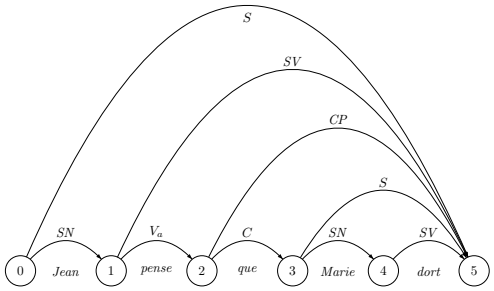
2. *Soit la grammaire* $S \rightarrow SN\,SV$; $SV \rightarrow V_a\,CP$; $CP \rightarrow C\,S$
$C \rightarrow que$; $V_a \rightarrow croit \mid pense$; $SN \rightarrow Jean \mid Marie$; $SV \rightarrow dort \mid ronfle$

   (a) *Donner la table des sous-chaînes bien formées produite par un parsing CYK de la phrase* Jean pense que Marie dort.

> **Rappel**
> La tables de sous-chaînes bien formée permet de représenter de façon compacte l'analyse possible de toutes les sous-chaînes d'une chaîne donnée, par une grammaire donnée.
> La même information peut être présentée sous la forme d'un DAG ou d'une liste d'item.
> En principe, la table est indépendante de l'algorithme qui la remplit. Cependant, selon la « famille » d'algorithmes en jeu, le contenu de la table n'est pas exactement le même. Pour la famille d'algorithmes dite **CYK**, les cases contiennent des règles, voire simplement la partie gauche de règles. Pour la famille dite **earley**, on a affaire à des **règles pointées**, ce qui offre évidemment d'autres possibilités algorithmiques.
> Cet exercice portait sur des tables des sous-chaînes bien formées « à la » CYK.

La table des sous-chaînes bien formées est la suivante, avec une représentation possible en DAG (on peut aussi numéroter de 1 à 6).

5	S	SV	CP	S	SV
4				SN	
3			C		
2		V_a			
1	SN				
	Jean	*pense*	*que*	*Marie*	*dort*
	0	1	2	3	4



   (b) *Supposons que la grammaire soit sous une forme légèrement différente (mêmes règles lexicales) :* $S \rightarrow SN\,SV$ *Donner la table des sous-chaînes bien formées*
$SV \rightarrow V_a\,C\,S$ *pour la même phrase avec cette nouvelle grammaire.*

Bien sûr, au sens strict, il n'est pas possible de faire tourner l'algorithme CYK sur une telle grammaire. Mais il est possible de proposer la table des sous-chaînes bien formées « à la CYK » (cette fois-ci, pour des raisons de clarté, les règles sont données en entier).

5	$S \rightarrow SN\,SV$	$SV \rightarrow V_a\,C\,S$		$S \rightarrow SN\,SV$	$SV \rightarrow dort$
4				$SN \rightarrow Marie$	
3			$C \rightarrow que$		
2		$V_a \rightarrow pense$			
1	$SN \rightarrow Jean$				
	Jean	*pense*	*que*	*Marie*	*dort*
	0	1	2	3	4

(c) *Proposer une version révisée de l'algorithme CYK qui accepterait de considérer des règles ayant 3 symboles non terminaux en partie droite.*

L'adaptation demandée ne doit pas aller jusqu'à considérer que **toutes** les règles (non lexicales) ont 3 symboles : il faut donc prévoir le cas habituel (partie droite à 2 symboles) et le cas nouveau (partie droite à trois symboles). Ci dessous une version de l'algorithme initial (indices grossièrement corrects), avec en rouge ce qu'il faut ajouter.

```
pour l (longueur du segment) de 2 à n :
   pour i (début du segment dans u) de 1 à n-l :
      pour k (coupure du segment en deux) de i+1 à i+l-1:
         si B in T[i, k+1] et C in T[k,i+l+1]
            et A -> BC in P:
               T[i, i+l+1] U= {A -> BC}
      pour k1 (coupure_1 du segment en trois) de i+1 à i+l-1:
         pour k2 (coupure_2 du segment en trois) de k1+1 à i+l-1:
            si B in T[i, k1+1] et C in T[k1,k2] et D in T[k2+1,I+L+1]
               et A -> BCD in P:
                  T[i, i+l+1] U= {A -> BCD}
```

(d) *Quel coût en complexité est introduit par une telle modification ?*

L'algorithme initial est de complexité en $O(n^3)$ (en négligeant le facteur lié à la taille du vocabulaire non terminal). Avec cette modification, on ajoute un niveau d'enchâssement, ce qui conduit à une complexité d'ordre n^4.

Commentaires : *On notera le coût très élevé d'une modification somme toute mineure ; on observera aussi que si on veut généraliser l'opération en autorisant certaines règles à avoir un nombre quelconque de symboles à droite, cette piste mène à une explosion combinatoire.*

3. *Soit la grammaire suivante qui reconnaît les expressions arithmétiques préfixées binaires (deux opérandes pour chaque opérateur).* $E \to D \mid (+\ E\ E) \mid (\times\ E\ E)$
$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

(a) *Donner l'arbre syntaxique correspondant à l'expression* $(+5(\times 63))$

Voir la réponse à la question (c).

(b) *Ajouter à cette grammaire des « actions sémantiques » et un attribut qui permette de calculer la valeur de l'expression analysée.*

Notation : un attribut nommé α, indices interprétés de la façon habituelle.

Règle	Action sémantique
D $\to$ 0	$\alpha_0 = 0$
D $\to$ 1	$\alpha_0 = 1$
$\vdots$	$\vdots$
D $\to$ 9	$\alpha_0 = 9$
E $\to$ D	$\alpha_0 = \alpha_1$
E $\to$ (+ E E)	$\alpha_0 = \alpha_3 + \alpha_4$
E $\to$ ($\times$ E E)	$\alpha_0 = \alpha_3 * \alpha_4$

(c) *Illustrer le calcul sur l'arbre syntaxique de la question (3a).*

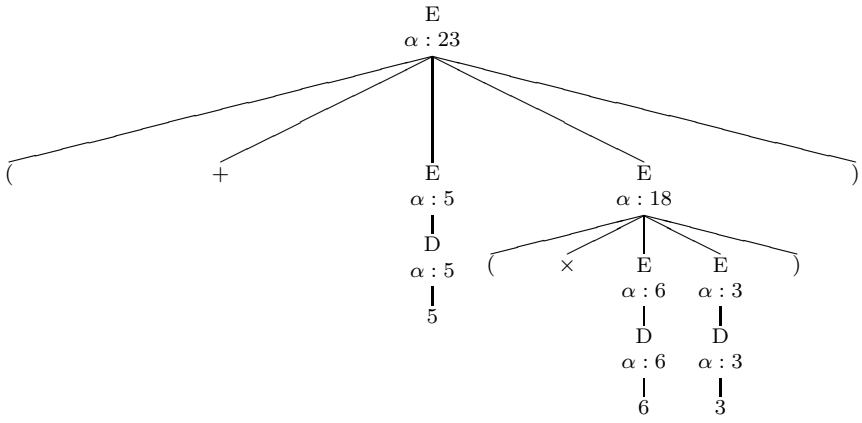
Voir page suivante.

On généralise la grammaire pour reconnaître des expressions arithmétiques impliquant un nombre quelconque (≥ 2) *d'opérandes (de sorte que l'expression* $(\times 5(+69)7)$ *soit bien formée) :*

$E \to D \mid (+\ E\ E') \mid (\times\ E\ E')$
$E' \to E'\ E \mid E$
$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

(d) *Proposer dans ce cas aussi une version « attribuée » de cette grammaire qui calcule la valeur de l'expression analysée.*

On rencontre une difficulté dans ce cas, à cause des règles sémantiques à associer aux règles E′ qui engendrent une liste d'un nombre quelconque d'opérandes. La stratégie



la plus simple consiste à former une liste avec les valeurs d'attribut de ces opérandes, laquelle liste sera exploitée au moment où l'opération sera connue. Pour simplifier (!), on suppose que l'attribut α contient selon les cas une valeur entière (quand il est associé au non terminal D, ou E) ou une valeur de type liste d'entiers (quand il est associé à E′). Cela donne, avec une notation que j'espère transparente :

Règle	Action sémantique	
$E \to D$	$\alpha_0 = \alpha_1$	
$E' \to E$	$\alpha_0 = \{\alpha_1\}$	*on crée une liste*
$E' \to E'\ E$	$\alpha_0 = \alpha_1.append(\alpha_2)$	*α_1 est une liste, α_2 un entier*
$E \to (\ +\ E\ E'\ )$	$\alpha_0 = \alpha_3$	*la valeur pour E est de nouveau un entier*
	for x in α_4 :	
	$\quad \alpha_0 = \alpha_0$ + x	
$E \to (\ \times\ E\ E'\ )$	$\alpha_0 = \alpha_3$	
	for x in α_4 :	
	$\quad \alpha_0 = \alpha_0$ * x	

Commentaires : *Il y avait de nombreuses autres solutions plus élégantes, par exemple en utilisant plusieurs attributs (l'un pour la valeur, l'autre pour la liste ; ou encore un pour la liste et un pour l'opération, ou encore des attributs booléens pour distinguer les différents cas). Attention toutefois avec les attributs hérités : ils ne peuvent pas être calculés* pendant *le parsing ascendant.*

Une dernière option peut être évoquée, même si elle conduit à une grammaire pathologiquement ambigüe : distinguer au niveau de la syntaxe les listes d'opérandes à additionner et les listes d'opérandes à multiplier :

Règle	Action sémantique
$E \to D$	$\alpha_0 = \alpha_1$
$E_1 \to E$	$\alpha_0 = \alpha_1$
$E_1 \to E\ E_1$	$\alpha_0 = \alpha_1 + \alpha_2$
$E_2 \to E$	$\alpha_0 = \alpha_1$
$E_2 \to E\ E_2$	$\alpha_0 = \alpha_1 * \alpha_2$
$E \to (\ +\ E\ E_1\ )$	$\alpha_0 = \alpha_3 + \alpha_4$
$E \to (\ \times\ E\ E_2\ )$	$\alpha_0 = \alpha_3 * \alpha_4$

(e) *Quel est le rôle des parenthèses dans les deux grammaires ?*

Question dépendante des précédentes. Les parenthèses sont généralement introduites pour lever les ambiguïtés, en particulier dans les expressions arithmétiques infixes. Mais en notation postfixe ou préfixe, comme ici, les parenthèses sont inutiles si le nombre d'opérandes est fixe. Dans le premier langage, on pourrait donc sans conséquences se passer des parenthèse. En revanche, dans le second langage, le nombre variables de opérandes rend les parenthèses nécessaires pour désambiguïser les expressions.