Polytech Marseille

Automates, Langages et Compilation
Lecture 6

N. Baudru

Informatique - 1e année

Table des matières

13 Analyseur syntaxique LL(1)

L'étude des langages réguliers, et en particulier des automates finis déterministes, nous a permis de décrire une méthode systématique pour construire un analyseur lexical reconnaissant les mots d'un lexique spécifiés par une expression régulière. Plus précisément, nous avons vu que, à chaque nouvel appel, l'analyseur lexical retourne le prochain token (lexème) du texte analysé (ou une erreur le cas échéant). La séquence de tokens ainsi produite forme le mot d'entrée analysé par l'analyseur syntaxique. En ce basant sur une grammaire définissant la syntaxe du langage, i.e. les bonnes séquences de tokens, l'analyseur syntaxique va déterminer si le mot lu en entrée est correct. Si c'est le cas, l'analyseur peut construit un arbre syntaxique qui sera utile pour les phases de compilation suivantes. Nous invitons le lecteur à relire la première partie de la lecture 1 afin de bien situer l'analyse syntaxique dans le processus de compilation. En particulier, il faut garder à l'esprit que les symboles terminaux des grammaires représentent maintenant des tokens.

Dans ce cours, nous allons présenter comment construire de manière systématique des analyseurs syntaxiques à partir de grammaires. En pratique, l'expressivité des grammaires hors-contextes est suffisante pour répondre à la plupart des besoins en compilation. Il faudra même se restreindre à certaines sous-classes car de nombreuses grammaires hors-contextes sont inadaptées à l'analyse syntaxique, principalement parce qu'elles peuvent être ambiguës ou non-déterministes.

Ambiguïté d'une grammaire Il arrive qu'un mot puisse être dérivé de plusieurs manières à partir de l'axiome d'une grammaire donnée. Si la façon de dériver un mot a un impact sur la sémantique sous-jacente du mot, alors la grammaire est ambiguë. La grammaire française est par exemple ambiguë, comme l'illustre la phrase suivante (source Wikipédia) : « Elle emporte les clefs de la maison au garage ». Comment faut-il comprendre cette phrase ? Elle emporte « les clefs de la maison » au garage, ou elle emporte les clefs « de la maison au garage » ? Les grammaires ambiguës ne se prêtent pas à l'analyse syntaxique car une séquence de tokens peut être analysée de plusieurs manières différentes, chacune d'elles produisant un arbre syntaxique différent. Quel arbre choisir pour poursuivre le processus de compilation ?

Analyseur déterministe Un analyseur syntaxique est *déterministe* s'il peut être décrit par un automate à pile déterministe. Il bénéficie ainsi d'un processus d'analyse efficace, sans backtracking. Au contraire, le processus d'analyse d'un automate à pile non-déterministe peut prendre un temps exponentiel dans la taille du mot à analyser. On sait déjà que certaines grammaires hors-contextes, dites non-déterministes, n'admettent pas d'automates à pile déterministes équivalents et sont donc peu adaptées à l'analyse syntaxique.

13.1 Ambiguïté des grammaires hors-contextes

Nous introduisons la notion d'ambiguïté par d'un exemple.

Nous supposons avoir déjà construit un analyseur lexical qui renvoie le token id lorsqu'une variable est reconnue, le token cte lorsqu'une constante est reconnue, et les tokens $+$, $\times$, (ou) lorsque un « plus », un « fois », une parenthèse ouvrante ou fermante sont reconnues. Nous définissons la syntaxe des « bonnes » expressions arithmétiques à l'aide d'une grammaire G utilisant les tokens id, cte, (et) comme symboles terminaux (cf. section 1 de la lecture 1). Cette grammaire a pour règles de production :

$$E \to id \mid cte \mid E + E \mid E \times E \mid (E).$$

Considérons maintenant l'expression

$$x + 2.5 \times 4 + (x + z).$$

Chaque appel à l'analyseur lexical renvoie le token suivant. Donc, après onze appels, nous obtenons la séquence de tokens suivante :

$$id + cte \times cte + (id + id).$$

Cette séquence de tokens correspond à un mot de la grammaire G. Il peut être dérivé à partir de l'axiome E de G de plusieurs manières selon la *stratégie de dérivation choisie*. Par exemple, une stratégie naturelle consiste à toujours dériver le non-terminal le plus à gauche (ou le plus à droite).

Dérivation gauche et droite Soit x un mot de $(V_T \cup V_N)^\star$, i.e. composé de symboles terminaux et non-terminaux. Soit $x \Rightarrow y$ une dérivation. Cette dérivation est appelée *dérivation à gauche* si elle consiste à remplacer le non-terminal le plus à gauche dans x. Similairement, la dérivation est appelée *dérivation à droite* si elle consiste à remplacer le non-terminal le plus à droite dans x.

Un mot y dérive (indirectement) à gauche (resp. droite) de x si il est obtenu à partir de x par des dérivations à gauche (resp. à droite) successives.

Le mot $id + cte \times cte + (id + id)$ peut être obtenu par une dérivation gauche :

$$\begin{aligned}
E &\to E + E \\
&\to E \times E + E \\
&\to E + E \times E + E \\
&\to id + E \times E + E \\
&\to id + cte \times E + E \\
&\to id + cte \times cte + E \\
&\to id + cte \times cte + (E) \\
&\to id + cte \times cte + (E + E) \\
&\to id + cte \times cte + (id + E) \\
&\to id + cte \times cte + (id + id)
\end{aligned}$$

Mais il peut aussi être obtenu par une dérivation droite :

$$\begin{aligned} E &\to E + E \\ &\to E + (E) \\ &\to E + (E + E) \\ &\to E + (E + id) \\ &\to E + (id + id) \\ &\to E \times E + (id + id) \\ &\to E \times cte + (id + id) \\ &\to E + E \times cte + (id + id) \\ &\to E + cte \times cte + (id + id) \\ &\to id + cte \times cte + (id + id) \end{aligned}$$

Ces séquences de dérivations peuvent être décrites par un *arbre syntaxique*. Cette nouvelle représentation est plus visuelle et met en valeur la façon dont le mot a été construit. C'est aussi la structure utilisée par le l'analyseur sémantique.

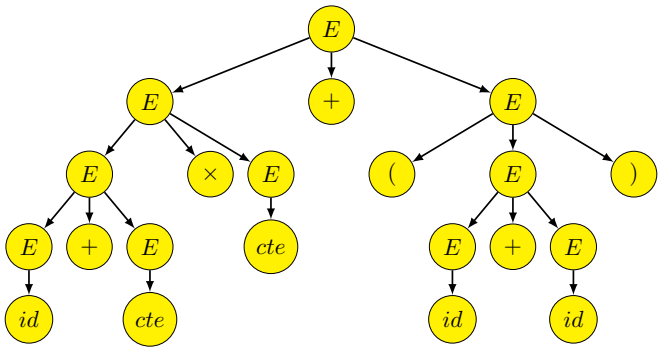
Arbre syntaxique

Définition 13.1 *Soit $G = (V_T, V_N, S, P)$ une grammaire hors-contexte.*
Un arbre syntaxique *de G est un arbre tel que*

- *tout noeud interne de l'arbre est étiqueté par un symbole non-terminal ;*
- *la racine de l'arbre est étiquetée par l'axiome S ;*
- *toute feuille de l'arbre est étiquetée par un symbole terminal ;*
- *si v est un noeud de l'arbre qui a pour fils $v_1, v_2, \ldots, v_k$ pris de gauche à droite, et si $v, v_1, \ldots, v_k$ sont étiquetés respectivement par $N, \alpha_1, \ldots, \alpha_k$, alors $N \to \alpha_1 \ldots \alpha_k$ est une règle de G.*

C'est un arbre pour le mot $u \in V_T^\star$ si les étiquettes des feuilles, lues de la gauche vers la droite, donnent le mot u.

A partir d'une suite de dérivations de G donnant le mot u, on peut construire un arbre syntaxique pour u. Par exemple, l'arbre syntaxique suivant correspond aux dérivations gauches et droites précédentes du mot $id + cte \times cte + (id + id)$. Un parcours en profondeur à gauche d'abord correspond à la dérivation gauche, alors qu'un parcours en profondeur à droite d'abord correspond à la dérivation droite.

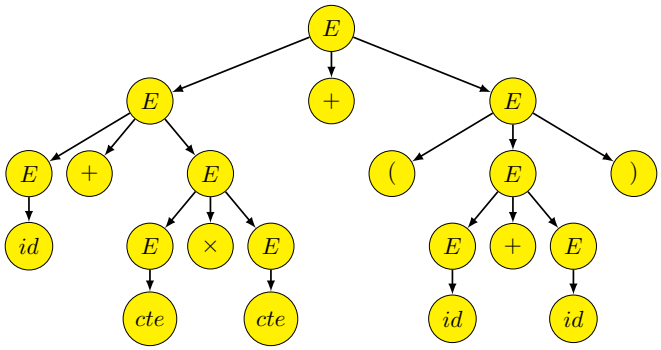


Les deux dérivations ayant le même arbre, elles sont considérées comme équivalentes : elles diffèrent par l'ordre dans lequel les non-terminaux sont réécrits, mais chaque non-terminal est réécrit en utilisant les mêmes règles dans les deux dérivations.

Il existe cependant, pour un même mot, des dérivations réellement différentes. Par exemple le mot $id + cte \times cte + (id + id)$ possède une autre séquence de dérivations droite :

$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E + (E) \\
&\rightarrow E + (E + E) \\
&\rightarrow E + (E + id) \\
&\rightarrow E + (id + id) \\
&\rightarrow E + E + (id + id) \\
&\rightarrow E + E \times E + (id + id) \\
&\rightarrow E + E \times cte + (id + id) \\
&\rightarrow E + cte \times cte + (id + id) \\
&\rightarrow id + cte \times cte + (id + id)
\end{aligned}
$$

Cette dernière a pour arbre syntaxique :



Grammaire et langage ambiguë L'exemple précédent montre qu'il peut exister plusieurs arbres syntaxiques d'une grammaire G pour un mot u. Dans ce cas la grammaire G est dite *ambiguë*. Un langage est *intrinsèquement ambigu* si toute grammaire qui le génère est ambiguë.

Notons ici que le choix de l'arbre syntaxique pour un mot donnée a un impact sur l'analyse sémantique. Par exemple, supposons que nous voulons évaluer l'expression arithmétique lors de l'analyse sémantique en effectuant les calculs lors d'un parcours infixe. Alors dans le première arbre, la multiplication n'est pas prioritaire sur l'addition, au contraire du second arbre.

Exercice 13.1 ☆ La grammaire suivante est-elle ambiguë ?

$$
P = \left| \begin{array}{l} S \rightarrow aSc \mid aS \mid T \\ T \rightarrow bTc \mid bT \mid \varepsilon \end{array} \right.
$$

Nous finissons avec un théorème montrant qu'il n'est pas possible de trouver une méthode systématique permettant de déterminer si une grammaire est ambiguë.

Théorème 13.1 *Déterminer si une grammaire est ambiguë est un problème indécidable.*

13.2 Grammaires LL(1)

Plusieurs classes de grammaires permettent de construire des analyseurs syntaxiques déterministes. Ces classes dépendent de la stratégie d'analyse recherchée : descendante, montante ou mixte. Dans une stratégie ascendante, on cherche à construire l'arbre syntaxique de la séquence de tokens en partant des feuilles vers la racine. A partir des symboles terminaux, on essaie de reconnaître des parties droites de règles pour les réduire aux symboles non-terminaux des parties gauches, et ainsi de suite jusqu'à arriver à l'axiome. Plusieurs classes de grammaires sont adaptées à ce type de stratégie comme les grammaires dites de précédences et les grammaires dites $LR(k)$.

A l'opposé, dans une stratégie descendante, on cherche à construire l'arbre syntaxique depuis la racine jusqu'aux feuilles en faisant des dérivations gauches. Les grammaires $LL(k)$ forment la plus large classe de grammaires permettant l'élaboration systématique d'analyseurs descendants déterministes. Le premier L signifie « left scan » et le second « left most derivation ». Une grammaire est $LL(k)$ s'il est possible de déterminer quelle règle doit être appliquée au non-terminal le plus à gauche au cours d'une dérivation par la connaissance de ce non-terminal et des k premiers symboles de la séquence restant à lire.

Seules les grammaires $LL(1)$ sont abordées dans ce cours.

Exemple 13.1 *Considérons la grammaire hors-contexte ayant, parmi d'autres, les deux règles de production suivantes :*

$$\begin{array}{l}\ldots\\ A \to Bbc \mid aBcbd\\ B \to \varepsilon \mid b \mid c\\ \ldots\end{array}$$

Supposons qu'au cours d'une dérivation le non-terminal le plus à gauche soit A. Nous voulons déterminer quelle règle appliquer à A pour obtenir le symbole courant lu. Si ce symbole est a alors nous n'avons pas d'autre choix que d'utiliser la règle $A \to aBcbd$. Si c'est un b ou un c, il faut choisir la règle $A \to Bbc$ car le B peut se dériver à son tour en b ou c. Tout autre symbole différent de a, b ou c ne pourra pas être obtenu et générera une erreur de syntaxe. En conclusion, si le non-terminal à gauche est A, il est possible de déterminer la règle à appliquer en comparant le symbole courant lu aux premiers symboles des mots dérivés en utilisant la règle $A \to Bbc$, ou aux premiers symboles des mots dérivés en utilisant la règle $A \to aBcbd$.

Supposons maintenant que le non-terminal le plus à gauche soit B et que le symbole courant lu soit un b. Ce symbole peut évidemment être obtenu en choisissant la règle $B \to b$. Mais ce n'est pas la seule possibilité. En effet, la séquence de dérivations $A \Rightarrow Bbc \Rightarrow bc$ montre que le b peut être obtenu en utilisant la règle $B \to \varepsilon$. Le b recherché correspond alors au b qui suit B dans la séquence de dérivations. Ce b a été obtenu plus tôt au cours du processus de dérivation. En conclusion, si le non-terminal à gauche est B, il n'est pas possible de déterminer la règle à appliquer en lisant un seul caractère : la grammaire n'est pas $LL(1)$.

Premiers et suivants L'exemple précédent montre que deux ensembles sont à considérer pour déterminer quelle règle appliquer au non-terminal X le plus à gauche : l'ensemble des symboles terminaux débutant les mots dérivés à partir de la partie droite u d'une règle $X \to u$, noté $premiers(u)$; et l'ensemble des symboles terminaux qui peuvent suivre immédiatement X au cours des différentes dérivations, noté $suivants(X)$.

Définition 13.2 *Soit G une grammaire hors-contexte ayant S comme axiome. Soit X un non-terminal de G et $u \in (V_T \cup V_N)^\star$ un mot constitué de symboles terminaux et non-terminaux.*

$$premiers(u) = \{a \in V_T \mid u \Rightarrow^* av\} \cup \{\varepsilon \text{ , si } u \Rightarrow^* \varepsilon\}$$

$$suivants(X) = \{a \in V_T \mid S \Rightarrow^* vXw \text{ et } a \in premiers(w)\}$$

L'ensemble $premiers(u)$ peut être inductivement calculé en utilisant les règles suivantes :

1. si $u = \varepsilon$, alors $premiers(u) = \{\varepsilon\}$
2. si $u = a \in V_T$, alors $premiers(u) = \{a\}$
3. si $u = X \in V_N$ et $X \to \alpha_1 \mid \ldots \mid \alpha_n$ alors $premiers(u) = \bigcup_{i\in\{1,\ldots,n\}} premiers(\alpha_i)$
4. si $u = xw$ avec $x \in V_T \cup V_N$ et $w \neq \varepsilon$, alors

$$premiers(u) = \begin{cases} premiers(x) & \text{si } \varepsilon \notin premiers(x) \\ (premiers(x) \setminus \{\varepsilon\}) \cup premiers(w) & \text{si } \varepsilon \in premiers(x) \end{cases}$$

Remarquez que $\varepsilon \in premiers(u)$ dès que $u \Rightarrow^* \varepsilon$.

Nous illustrons l'idée cachée derrière le point 4 par un exemple. Supposons que $u = BCaD$ avec a un terminal et B, C et D des symboles non-terminaux. Par quelles lettres terminales débutent les mots obtenus en dérivant u ? Autrement dit, que vaut $premiers(u)$? Clairement $premiers(u)$ contient les premières lettres des mots obtenus en dérivant B : $premiers(B) \subseteq premiers(u)$. Nous avons l'égalité si $\varepsilon \notin premiers(B)$. Mais si $\varepsilon \in premiers(B)$ alors B peut se dériver en ε, et par suite u se dérive en CaD. Il suit que $premiers(u)$ contient aussi les premières lettres des mots obtenus en dérivant CaD. Les premiers de CaD sont calculés de la même manière. On voit bien apparaître la règle 4 du calcul des premiers.

L'ensemble $suivants(X)$ peut être inductivement calculé en utilisant la règle suivante : si il existe une règle $B \to uAv$ (v peut être le mot vide ε), alors

$$suivants(A) \supseteq \begin{cases} premiers(v) & \text{si } \varepsilon \notin premiers(v) \\ (premiers(v) \setminus \{\varepsilon\}) \cup suivants(B) & \text{si } \varepsilon \in premiers(v) \end{cases}$$

Condition de Knuth Soit G un langage hors-contexte. Knuth a décrit des conditions nécessaires et suffisantes pour qu'une grammaire hors-contexte G soit $LL(1)$, et donc adaptée à l'analyse syntaxique descendante déterministe. Ces conditions sont :

1. G n'a pas de récursivité gauche, i.e. de dérivations du type $X \Rightarrow^* Xu$.
2. Pour chaque couple de règles $A \to \alpha_1$ et $A \to \alpha_2$ avec $\alpha_1 \neq \alpha_2$, nous avons

   $$possibles(A \to \alpha_1) \cap possibles(A \to \alpha_2) = \emptyset$$

   où $possibles(X \to \alpha)$ représente l'ensemble des symboles terminaux peuvant apparaître en prenant la règle $X \to \alpha$:

   $$possibles(X \to \alpha) = \begin{cases} premiers(\alpha) & \text{si } \varepsilon \notin premiers(\alpha) \\ (premiers(\alpha) \setminus \{\varepsilon\}) \cup suivants(X) & \text{si } \varepsilon \in premiers(\alpha) \end{cases}$$

La première condition évite que l'analyseur syntaxique boucle. En effet, un analyseur syntaxique descendant recherche un arbre syntaxique pour un mot u donné en effectuant des dérivations gauches à partir de l'axiome. En cas de présence d'une règle $X \to Xa$, ou plus généralement si $X \Rightarrow^* Xu$, le processus de dérivation ne s'arrêtera pas.

La deuxième condition vérifie qu'il est possible de déterminer quelle règle doit être appliquée au non-terminal le plus à gauche au cours d'une dérivation par la connaissance de ce non-terminal et du premier symbole de la séquence restant à lire. Les idées derrières cette deuxième condition ont déjà été présentées dans l'exemple 13.1.

Exemple 13.2 *Soit la grammaire G ayant S pour axiome et les règles de productions suivantes :*

$$\left| \begin{array}{l} S \to W\$ \\ W \to a \mid (X) \mid \varepsilon \\ X \to WY \\ Y \to \cdot WY \mid \varepsilon \end{array} \right.$$

Commençons par calculer les premiers et les suivants de chaque non-terminal.

	$premiers$	$suivants$
S	$a, (, \$$	
W	$a, (, \varepsilon$	$\$, \cdot,)$
X	$a, (, \cdot, \varepsilon$	$)$
Y	$\cdot, \varepsilon$	$)$

Quelques explications. Pour calculer les premiers de W, on regarde les règles ayant W dans la partie gauche. Ici il y en a trois : $W \to a|(X)|\varepsilon$. Les premiers de W sont les premiers de a, plus les premiers de (X) plus les premiers de ε, soit $\{a, (, \varepsilon\}$.

Pour X, il faut regarder la règle $X \to WY$. les premiers de X contiennent les premiers de W. Comme $\varepsilon \in premiers(W)$, il faut aussi considérer les premiers de Y. Au final, on obtient :

$$premiers(X) = (premiers(W) \setminus \{\varepsilon\}) \cup premiers(Y).$$

Pour calculer les suivants de W, il faut considérer les trois règles où W apparait à droite. Considérons d'abord $S \to W\$$. On apprend de cette règle que $\$$ est dans les suivants de W.

Considérons maintenant $X \to WY$. Les suivants de W contiennent les premiers de Y. Mais comme $\varepsilon \in premiers(Y)$, il faut aussi y ajouter les suivants de X Au final,

$$suivants(W) \supseteq (premiers(Y) \setminus \{\varepsilon\}) \cup suivants(X).$$

Enfin, un raisonnement similaire pour la règle $Y \to \cdot WY$ donne :

$$suivants(W) \supseteq (premiers(Y) \setminus \{\varepsilon\}) \cup suivants(Y).$$

Vérifions maintenant que G est $LL(1)$, i.e. vérifie les conditions de Knuth.

— *Il n'y a pas de récursivité gauche.*

— *Pour $S \to W\$$ et $X \to WY$, il n'a pas de problème car pas de choix possible.*

— *Pour $W \to a \mid (X) \mid \varepsilon$: les conditions sont respectées car*

$$\begin{aligned} possibles(W \to a) \cap possibles(W \to (X)) &= \{a\} \cap \{(\} = \emptyset \\ possibles(W \to a) \cap possibles(W \to \varepsilon) &= \{a\} \cap \{\$, \cdot,)\} = \emptyset \\ possibles(W \to (X)) \cap possibles(W \to \varepsilon) &= \{(\} \cap \{\$, \cdot,)\} = \emptyset. \end{aligned}$$

— *Les conditions sont aussi satisfaites pour $Y \to \cdot WY \mid \varepsilon$ car*

$$possibles(Y \to \cdot WY) \cap possibles(Y \to \varepsilon) = \{\cdot\} \cap \{)\} = \emptyset.$$

Exercice 13.2 ☆ Soit la grammaire $G = (\{S, A, M\}, \{a, b\}, S, P)$ ayant pour règles :

$$P = \left| \begin{array}{l} S \to A\$ \\ A \to aM \mid \varepsilon \\ M \to aAbb \mid \varepsilon \end{array} \right.$$

Déterminez les premiers et les suivants de S, A et M et montrez que G est $LL(1)$.

13.3 Des grammaires LL(1) aux analyseurs syntaxiques

Nous présentons dans cette section comment transformer systématiquement une grammaire $LL(1)$ en un analyseur syntaxique descendant déterministe reconnaissant le même langage. Cet analyseur peut être vu de manière équivalente comme un automate à pile déterministe, ou comme un programme constitué de procédures mutuellement récursives.

Comme pour les analyseurs lexicaux, l'automate ou le programme doit pouvoir déterminer si il a atteint la fin du mot à analyser. Pour cela, nous utilisons un symbole terminal spécial $\$$ de fin de mots (en pratique, $\$$ sera par exemple le caractère de fin de fichier EOF), et supposons dans ce qui suit que l'axiome S de notre grammaire est à l'origine d'une *unique* règle, de la forme $S \to A\$$ pour un certain $A \in V_N$. Ainsi, tout mot reconnu par la grammaire ou l'automate sera de la forme $u\$$. Nous avons donc affaire à un langage préfixe (voir section 12.3), qui est une condition nécessaire pour être reconnu par un automate à pile acceptant par pile vide. Notez que toute grammaire peut être étendu en ce sens.

Soit $G = (V_T, V_N, S, P)$ une grammaire hors contexte $LL(1)$ comme décrite ci-dessus.

Représentation par automate à pile déterministe Nous construisons à partir de G un automate à pile déterministe acceptant par pile vide. Cet automate possède un état q_a par symbole terminal a de $V_T \cup \{\$\}$, plus un état initial q. L'alphabet de pile est $V_T \cup \{\$\} \cup V_N$, avec l'axiome S comme symbole de fond de pile. Intuitivement, l'automate va essayer de déterminer si le mot $u\$$ lu en entrée peut être obtenu par une dérivation gauche de la grammaire :

— De l'état q, l'automate lit une lettre $a \in V_T \cup \{\$\}$ du mot en entrée et passe dans l'état q_a.

— A partir de l'état q_a, l'automate essaie de simuler dans la pile une séquence de dérivations gauches lui permettant d'obtenir a au sommet de la pile. Pour cela, lorsque X est au sommet de la pile, l'automate recherche une règle $X \to \alpha$ avec $a \in possibles(X \to \alpha)$. Si cette règle existe, il dépile X et empile α. Sinon, l'automate s'arrête : a ne peut pas être produite par la grammaire et donc le mot lu n'appartient pas au langage spécifié par G.

— Si a apparaît au sommet de la pile, l'automate dépile a est retourne en q. La lettre suivante du mot en entrée pourra être lue.

Formellement, nous avons pour chaque symbole terminal $a \in V_T \cup \{\$\}$:

— une transition $(q, \varepsilon, x, q_a, x)$ pour chaque $x \in V_T \cup V_N$;

— une transition $(q_a, \varepsilon, X, q_a, \alpha)$ pour chaque règle $X \to \alpha$ telle que $a \in possibles(X \to \alpha)$

— une transition $(q_a, \varepsilon, a, q, \varepsilon)$.

Comme nous avons supposé que S est à l'origine d'une unique règle du type $S \to A\$$, la pile deviendra vide seulement après l'exécution de la transition $(q_\$, \varepsilon, \$, q, \varepsilon)$. Notez aussi que la séquence des transitions effectuées pour accepter un mot $u\$$ simule en fait une dérivation gauche pour le mot $u\$$. C'est la raison pour laquelle cet automate décrit le même langage que la grammaire G. Enfin, l'arbre syntaxique correspondant à $u\$$ peut facilement être construit à partir de la dérivation gauche pour $u\$$.

Nous pourrions facilement vérifier que cet automate est déterministe car G est $LL(1)$.

Exemple 13.3 *Poursuivons avec la grammaire G de l'exemple 13.2. Nous avons déjà vu que cette grammaire est $LL(1)$. Nous représentons l'automate à pile correspondant par sa table prédictive M. Cette table indique, pour chaque symbole a (indiçant les colonnes), la transition que l'automate doit effectuer à partir de l'état correspondant q_a lorsque le symbole non terminal X (indiçant les lignes) se trouve au sommet de la pile. Autrement dit, $M[X, a] = \alpha$ représente la transition $(q_a, \varepsilon, X, q_a, \alpha)$ (qui elle-même simule la règle $X \to \alpha$). Notez que préciser ces transitions suffit pour décrire parfaitement l'automate car les autres transitions (de q vers q_a d'une part, et de q_a vers q d'autre part) sont toujours identiques.*

Ci-dessous, nous supposons que les règles de la grammaire G sont numérotées dans l'ordre croissant. Ce sont ces numéros que apparaissent dans la table et dans le calcul qui suivent.

	a	$($	$)$	$\cdot$	$\$$
S	$W\$, 0$	$W\$, 0$			$W\$, 0$
W	$a, 1$	$(X), 2$	$\varepsilon, 3$	$\varepsilon, 3$	$\varepsilon, 3$
X	$WY, 4$	$WY, 4$	$WY, 4$	$WY, 4$	
Y			$\varepsilon, 6$	$\cdot WY, 5$	

Analysons le mot « $(a\cdot)$ » à la manière d'un automate à pile (le contenu de la pile est en gras) :

$$
\begin{array}{ccccccccc}
(a.)\$, q, \boldsymbol{S} & \vdash & a.)\$, q_(, \boldsymbol{S} & \overset{0}{\vdash} & a.)\$, q_(, \boldsymbol{W\$} & \overset{2}{\vdash} & a.)\$, q_(, \boldsymbol{(X)\$} & & \\
\overset{shift}{\vdash} & a.)\$, q, \boldsymbol{X)\$} & \vdash & .)\$, q_a, \boldsymbol{X)} & \overset{4}{\vdash} & .)\$, q_a, \boldsymbol{WY)\$} & \overset{1}{\vdash} & .)\$, q_a, \boldsymbol{aY)\$} & \\
 & \overset{shift}{\vdash} & .)\$, q, \boldsymbol{Y)\$} & \overset{shift}{\vdash} &)\$, q., \boldsymbol{Y)\$} & \overset{5}{\vdash} &)\$, q., \boldsymbol{.WY)\$} & & \\
 & \overset{shift}{\vdash} &)\$, q, \boldsymbol{WY)\$} & \overset{shift}{\vdash} & \$, q_), \boldsymbol{WY)\$} & \overset{3}{\vdash} &)\$, q_), \boldsymbol{Y)\$} & \overset{6}{\vdash} &)\$, \boldsymbol{)\$} \\
 & & \overset{shift}{\vdash} & \$, q, \boldsymbol{\$} & \vdash & \varepsilon, q_\$, \boldsymbol{\$} & \overset{success}{\vdash} & \varepsilon, q, \varepsilon &
\end{array}
$$

Le mot « $(a\cdot)$ » est reconnu par l'automate. La dérivation gauche correspondante (et donc l'arbre syntaxique) est donnée par la séquence des numéros de règles : 0241536.

Représentation par un programme récursif L'analyseur syntaxique peut aussi être décrit par un programme constitué de plusieurs prodédures mutuellement récursives. Intuitivement, la gestion en mémoire des appels récursifs va faire office de pile.

Le programme de l'analyseur est composé d'une procédure `parseA()` par non-terminal A de la grammaire. Le point d'entrée du programme est la méthode correspondante à l'axiome. Le programme utilise deux variables globales : `currentSymbol` correspond au symbole courant lu, et `trace` enregistre l'historique des règles appliquées (leur numéro pour être exact). Une fonction `nextSymbol()` enregistre dans `currentSymbol` le symbole suivant dans le mot en cours d'analyse. Une fonction `addTrace(i)` ajoute l'entier i à droite de la liste `trace`.

Trois principes généraux doivent guider l'écriture d'une procédure `parseA()` correspondant à un non-terminal A avec $A \to \alpha_1 \mid \cdots \mid \alpha_n$:

- Le symbole courant est toujours connu au moment où la procédure est appelée ;
- La procédure contient un bloc d'instructions pour chaque α_i. Elle utilise les ensembles $possibles(A \to \alpha_i)$ pour déterminer quel bloc d'instructions exécuter (i.e. quelle règle choisir) en fonction de la valeur de `currentSymbol`.
- Le bloc d'instructions d'un α_i contient une instruction par symbole de α_i. Chaque non-terminal B de α_i est traduit en l'instruction `parseB();`. Chaque symbole terminal a de α_i est traduit en l'instruction `if (currentSymbol == 'a') nextSymbol(); else ERROR;`, où ERROR est une instruction arrêtant l'analyse syntaxique.

Précisons aussi que la procédure correspondant à l'axiome à un comportement légèrement différent : elle doit en plus initialiser les variables globales et déterminer si l'analyse syntaxique est un succès.

Exemple 13.4 *Reprenons la grammaire LL(1) de l'exemple 13.2. La procédure `parseS()` correspond à l'axiome $S \to W\$$. Elle initialise le processus d'analyse en lisant le premier symbole. Si après l'analyse de W le symbole courant est le symbole $ alors l'analyse a réussi.*

```
void parseS()  {
    nextSymbol();
    trace = null;

    if ( currentSymbol in possibles(S -> W$) ) {
        addTrace(0);
        parseW();
        if (currentSymbol == '$') SUCCESS; else ERROR;
    }
    else ERROR
}
```

```
parseW() {
    if ( currentSymbol in possibles(W -> a) ) {
        addTrace(1);
        if (currentSymbol == 'a') nextSymbol();  else ERROR;
    }
    else if ( currentSymbol in possibles(W -> (X)) ) {
        addTrace(2);
        if (currentSymbol == '(') nextSymbol(); else ERROR;
        parseX();
        if (currentSymbol == ')') nextSymbol(); else ERROR;
    }
    else if ( currentSymbol in possibles( W -> epsilon ) ) {
        addTrace(3);
    }
    else ERROR
}
```

```
parseX() {
    if ( currentSymbol in possibles(X -> WY) ) {
        addTrace(4);
        parseW();
        parseY();
    else ERROR
}
```

```
parseY() {
    if ( currentSymbol in possibles(Y -> .WY) ) {
        addTrace(5);
        if (currentSymbol == '.') nextSymbol(); else ERROR;
        parseW();
        parseY();
    }
    else if ( currentSymbol in possibles(Y -> epsilon) ) {
        addTrace(6);
    }
    else ERROR
}
```

Lier l'analyse lexicale et l'analyse syntaxique Nous rappelons que les symboles terminaux utilisés lors de l'analyse syntaxique sont en fait les tokens fournis par l'analyseur lexical. La fonction `bool scanner()` de la section 7.4 peut être étendu pour retourner un token lorsque l'analyse lexicale réussit, par exemple <id> lorsqu'un identifiant est reconnu, ou <nb> lorsque c'est un nombre.

Dans ce cas, la variable `currentSymbol` doit être partagée par les fonctions `scanner()` et `parser()`. La fonction `scanner()` stocke le token trouvé dans la variable `currentSymbol`. La fonction `parser()` fait un appel à `scanner()` pour obtenir le prochain token, en lieu et place de la fonction `nextSymbol()`.

Exercice 13.3 ☆ Construisez la table prédictive et le programme récursif correspondant à la grammaire de l'exercice 13.2.

Exercice 13.4 ☆☆ Soit la grammaire $G = (\{A, B, C, D\}, \{a, b, (,), [,], ;\}, A, P)$ avec A pour axiome et P l'ensemble des règles de productions suivantes :

$S \to A\$$
$A \to (B) \mid \varepsilon$
$B \to C; \mid aA$
$C \to DA[C] \mid \varepsilon$
$D \to b \mid \varepsilon$

1. Donnez l'arbre de dérivation gauche correspondant au mot $(a(b[];))$.
2. Calculez les premiers et les suivants de chaque non-terminal.
3. Vérifiez que les conditions LL(1) (les conditions de Knuth) sont satisfaites pour G.
4. Écrivez dans un langage algorithmique un analyseur descendant récursif pour G.