Analyse Syntaxique – CTD - Partie 3

Automates à pile

Un *automate à pile* $\mathcal{A}$ est donné par un tuple $(\Sigma, \mathcal{Z}, z_\perp, Q, q_i, F, \Delta)$ où :
- Σ est un *alphabet d'entrée* fini (les terminaux),
- $\mathcal{Z}$ est un *alphabet de pile* fini,
- $z_\perp \in \mathcal{Z}$ est le *symbole initial de pile*,
- Q est un *ensemble fini d'états*,
- $q_i \in Q$ est l'*état initial* et $F \subseteq Q$ est l'ensemble des *états finaux*,
- Δ est une *relation de transition*

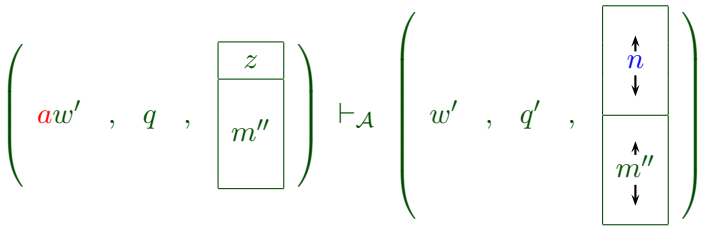
$$\Delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \mathcal{Z} \times Q \times \mathcal{Z}^*)$$

Transitions Un élément $(q, y, z_1, q', z_2z_3z_4)$ de Δ avec
- $q, q' \in Q$,
- $y \in \Sigma \cup \{\epsilon\}$,
- $z_1, z_2, z_3, z_4 \in \mathcal{Z}$

est appelé *transition* et est souvent noté : $q, y, z_1 \to q', z_2z_3z_4$.
Si $y \in \Sigma$, on parle de Σ-transition et si $y = \epsilon$ d'ϵ-transition.
Intuitivement, "si on est dans l'état q, que le sommet de pile est z_1 et qu'on peut lire y en tête du mot, alors
- on lit y et on dépile z_1,
- on passe dans l'état q',
- et on empile z_2, z_3, puis z_4.

Configuration La *configuration* d'un automate à pile $\mathcal{A}$ est donnée par :
- un mot $w \in \Sigma^*$
- un état "courant" q,
- un contenu de la pile $\begin{array}{|c|}\hline z_a \\ z_2 \\ z_b \\ z_a \\ \hline \end{array}$

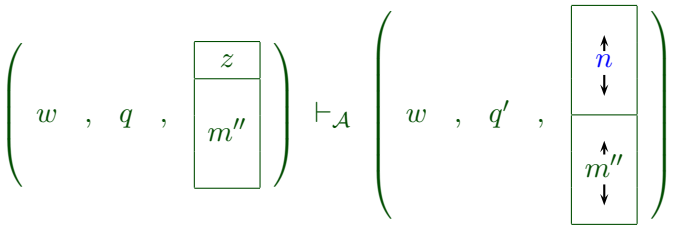
Une *configuration* c d'un automate à pile $(\Sigma, \mathcal{Z}, z_\perp, Q, q_i, F, \Delta)$ est un élément de $\Sigma^* \times Q \times \mathcal{Z}^*$.
Le mot de $\mathcal{Z}^*$ est le contenu de la pile **lu du bas de pile vers le haut de pile** $(z_az_bz_2z_a)$.
Configuration et Σ-transition
L'automate $\mathcal{A}$ réalise une Σ-transition d'une configuration $c = (w, q, m)$ vers une configuration $c' = (w', q', m')$ (noté $c \vdash_{\mathcal{A}} c'$) si :
- il existe une transition $q, a, z \to q', n \in \Delta$,
- $w = aw'$, $m = m''z$,
- $m' = m''n$.



Configuration et ϵ-transition

L'automate $\mathcal{A}$ réalise une ϵ-transition d'une configuration $c = (w, q, m)$ vers une configuration $c' = (w', q', m')$ (noté $c \vdash_{\mathcal{A}} c'$) si :

- il existe une transition $q, \epsilon, z \to q', n \in \Delta$,
- $w = w'$, $m = m''z$,
- $m' = m''n$.



Acceptation par état final

Pour un automate à pile $\mathcal{A}$, on note $\vdash^*_{\mathcal{A}}$ la clôture réflexive et transitive de $\vdash_{\mathcal{A}}$.
Un mot $w \in \Sigma^*$ est *accepté par état final* par un automate à pile $\mathcal{A} = (\Sigma, \mathcal{Z}, z_\perp, Q, q_i, F, \Delta)$ si pour la configuration $(w, q_i, z_\perp)$, il existe un état $q_f \in F$ et un mot $m \in \mathcal{Z}^*$ tel que

$$(w, q_i, z_\perp) \vdash^*_{\mathcal{A}} (\epsilon, q_f, m)$$

Le *langage accepté par état final par un automate à pile* est l'ensemble des mots acceptés par cet automate.

$$L^F(\mathcal{A}) = \{w \in \Sigma^* \mid (w, q_i, z_\perp) \vdash^*_{\mathcal{A}} (\epsilon, q_f, m)\}$$

Acceptation par pile vide

Un mot $w \in \Sigma^*$ est *accepté par pile vide* par un automate à pile $\mathcal{A} = (\Sigma, \mathcal{Z}, z_\perp, Q, q_i, F, \Delta)$ si pour la configuration $(w, q_i, z_\perp)$, il existe un état q tel que

$$(w, q_i, z_\perp) \vdash^*_{\mathcal{A}} (\epsilon, q, \epsilon)$$

Le *langage accepté par pile vide par un automate à pile* est l'ensemble des mots acceptés par cet automate.

$$L^S(\mathcal{A}) = \{w \in \Sigma^* \mid (w, q_i, z_\perp) \vdash^*_{\mathcal{A}} (\epsilon, q, \epsilon)\}$$

Equivalences des acceptations

Soit L un langage accepté par état final par un automate $\mathcal{A}$, alors il existe un automate à pile $\mathcal{A}'$ tel que le langage $L^S(\mathcal{A}')$ accepté par $\mathcal{A}'$ par pile vide vérifie $L = L^S(\mathcal{A}')$.
Soit L un langage accepté par pile vide par un automate $\mathcal{A}$, alors il existe un automate à pile $\mathcal{A}'$ tel que le langage $L^F(\mathcal{A}')$ accepté par $\mathcal{A}'$ par état final vérifie $L = L^F(\mathcal{A}')$.

Automates à pile et langages algébriques

Théorème Si L est un langage algébrique alors il existe un automate à pile $\mathcal{A}$ (acceptant par pile vide) tel que $L = L^S(\mathcal{A})$.

Principe de la preuve : Si L est un langage algébrique alors il existe une grammaire $G = (\Sigma, V, S, P)$ qui engendre L. On peut de plus supposer que toutes les règles de production de P sont de la forme[1] $X \to a$ ($a \in \Sigma$) ou $X \to \beta$ ($\beta \in V^*$). Pour la grammaire $G = (\Sigma, V, S, P)$, on construit l'automate à pile $\mathcal{A}_L$ acceptant par pile vide $\mathcal{A}_L = (\Sigma, \mathcal{Z}, z_\perp, Q, q_i, F, \Delta)$ tel que

- les terminaux de la grammaire constituent l'alphabet d'entrée,
- $\mathcal{Z} = V$: l'alphabet de pile est constitué des non-terminaux,
- $z_\perp$ est l'axiome S,
- $Q = \{q_i\}$ et $F = \emptyset$,
- $\Delta = \{(q_i, a, X) \to (q_i, \epsilon) \mid X \to a \in P\} \cup$
  $\{(q_i, \epsilon, X) \to (q_i, \beta^R) \mid X \to \beta \in P \text{ et } \beta \in V^*\}$
  où β^R est le miroir de β (β lu de droite à gauche).

On peut montrer que $L^S(\mathcal{A}_L) = L(G) = L$.

En fait il existe deux manières de construire un automate à pile pour une grammaire donnée. Cette première méthode empile les membres droits de règles et utilise le mot d'entrée pour dépiler. C'est une méthode descendante.

Une seconde méthode empile le mot d'entrée et utilise les règles de la grammaire pour dépiler. C'est une méthode ascendante. Il suffit de modifier l'automate à pile ci-dessus :

- $z_\perp = \epsilon$: la pile est vide au début,
- $\Delta = \{(q_i, a, \epsilon) \to (q_i, a) \mid X \to a \in P\} \cup$
  $\{(q_i, \epsilon, \beta) \to (q_i, X) \mid X \to \beta \in P \text{ et } \beta \in V^*\}$
- on doit terminer avec l'axiome S en sommet de pile, on ajoute donc à Δ la transition $(q_i, \epsilon, S) \to (q_i, \epsilon)$.

Théorème Pour tout automate à pile $\mathcal{A}$ (acceptant par pile vide), le langage $L^S(\mathcal{A})$ est un langage algébrique.

Automates à pile déterministes

Un automate à pile $\mathcal{A} = (\Sigma, \mathcal{Z}, z_\perp, Q, q_i, F, \Delta)$ acceptant par état final est *déterministe* si l'ensemble Δ des règles de transitions vérifie : pour toute paire de règles $(q, x, z) \to (q_1, m)$ et $(q, x', z) \to (q'_1, m')$ de Δ,

- si $x = x'$ alors $q_1 = q'_1$ et $m = m'$,
- si $x \in \Sigma$ alors $x' \in \Sigma$ (i.e. $x' \neq \epsilon$).

Un langage algébrique L est *déterministe* s'il existe un automate à pile $\mathcal{A}$ (acceptant par état final) *déterministe* tel que $L^F(\mathcal{A}) = L$.

Tous les langages algébriques ne sont pas déterministes : $\{w \in (a+b)^* \mid w \text{ est un palindrome}\}$ n'est pas déterministe.

$\Rightarrow$ Les automates à pile ne sont pas "déterminisables".

1. Si les règles ne sont pas de cette forme alors on peut transformer la grammaire de la façon suivante : pour chaque terminal a dans β, on crée un nouveau non-terminal X_a, on remplace toutes les occurrences de a par X_a dans β et on ajoute la production $X_a \to a$.

Analyse syntaxique

L'analyse syntaxique est avant tout le problème d'appartenance des grammaires algébriques

Définition Le *problème d'appartenance* est, donnés une grammaire algébrique G et un mot m de Σ^*, de décider si $m \in L(G)$.

Une méthode naïve consiste à génèrer toutes les dérivations possibles partant de l'axiome (ou de manière équivalente, itérativement le langage engendré étendu) ;
- Si on arrive à engendrer le mot m alors on retourne OUI
- sinon on retourne NON

Exemple pour les expressions régulières : $G = (\{a, b, (,), +, *\}, \{R\}, R, P)$ avec $P = \{R \to R{+}R \mid RR \mid (R) \mid R* \mid a \mid b\}$
- on génère les "langages étendus" partiels L_i^*

$L_0^* = \{R\}, \quad L_1^* = L_0^* \cup \{a, b, R+R, RR, R*, (R), \}$

$$L_2^* = L_1^* \cup \left\{ \begin{array}{l} a+R, b+R, R+a, R+b, aR, Ra, bR, Rb, \\ a*, b*, (a), (b), R+R+R, R+RR, \\ RR+R, R+R*, (R+R), .. \end{array} \right\}$$

$\ldots$
- A chaque étape i, on teste si $m \in L_i^*$.

Problème : pour certaines grammaires, quand doit-on s'arrêter en répondant NON ?

$G' = (\{a\}, \{A, B, R\}, A, \{A \to AB \mid R,\ B \to \epsilon,\ R \to aR \mid \epsilon\})$

Proposition Il existe un algorithme qui, pour toute grammaire $G = (\Sigma, V, S, P)$ et pour tout mot m de Σ^*, teste si $m \in L(G)$.
Si $m = \epsilon$ alors
- on calcule $V_\epsilon = \{A \in V \mid A \to_G^* \epsilon\}$;
- on retourne $S \in V_\epsilon$;

Sinon
- on transforme G en une grammaire réduite et propre G'.
- on utilise l'algorithme naïf vu précédemment pour générer $L_{|m|}^*$ pour la grammaire G'
- on retourne $m \in L_{|m|}^*$

Complexité : A chaque étape, on a n possibilités de règles de la grammaire et on a au moins $|m|$ étapes.

dans le pire des cas : $O(n^{|m|})$

Analyseurs syntaxiques
Les algorithmes d'analyse syntaxique sont en général scindés en deux familles :
- *les analyseurs descendants* : partant de l'axiome, ils tentent de construire un arbre de dérivation (ou un dérivation droite/gauche) pour le mot à analyser. La construction de l'arbre se fait de haut en bas. Les règles de production sont utilisées de la gauche vers la droite. `JavaCC` génère une analyse récursive descendante.
- *les analyseurs ascendants* : partant du mot à analyser, ils tentent de construire un arbre de dérivation (ou un dérivation droite/gauche) dont la racine est l'axiome. La construction de l'arbre se fait de bas en haut. Les règles de production sont utilisées de la droite vers la gauche. `Yacc` et `CUP` génèrent une analyse ascendante.

Les *analyseurs syntaxiques "universels"* fonctionnent pour toute grammaire algébrique :
- *algorithme de Cocke-Younger-Kasami* pour des grammaires en forme normale de Chomsky ($\simeq$ Greibach) ;
- *algorithme de Earley* pour des grammaires algébriques quelconques.

Leur complexité est en $o(n^3)$, où n est la longueur du mot à analyser.
- cette complexité est trop élevée (programmes de grande taille), on souhaite $o(n)$.
- Les langages de programmation peuvent être décrits par des grammaires en général peu complexes (en particulier, non-ambiguës).

On peut réaliser une *analyse syntaxique avec rebroussement* :
- une *méthode descendante* qui construit une dérivation gauche
- on utilise autant que possible le mot d'entrée (dans une lecture de gauche a droite) pour déterminer la règle à utiliser.
- si on ne peut pas déterminer la règle à utiliser, on fait un choix ....
- si le choix s'avère mauvais, on *"rebrousse son chemin"* pour faire un autre choix.

Exemple d'acceptation sans rebroussement avec
$G = (\{a, b, c, d\}, \{S, T\}, S, \{S \to aSbT \mid cT \mid d,\ T \to aT \mid bS \mid c\})$

$$\begin{array}{cc} a\ c\ c\ b\ b\ a\ d\ b\ c & a\ c\ c\ b\ b\ a\ d\ b\ c \\ \triangle & \triangle \\ S & S \xrightarrow{g}_G aSbT \end{array}$$

$$\begin{array}{c} a\ c\ c\ b\ b\ a\ d\ b\ c \\ \triangle \end{array}$$

$$\ldots \xrightarrow{g}_G acTbT \xrightarrow{g}_G accbT \xrightarrow{g}_G accbbS \xrightarrow{g}_G accbbaSbT \xrightarrow{g}_G accbbadbc$$

Le mot est accepté.
Exemple avec rebroussement avec $G' = (\{a, b, c\}, \{S, A\}, S, \{S \to aAb,\ A \to cd \mid c\})$

$$\begin{array}{ccc} a\ c\ b & a\ c\ b & a\ c\ b \\ \triangle & \triangle & \triangle \\ S & S \xrightarrow{g}_{G'} aAb & S \xrightarrow{g}_{G'} aAb \xrightarrow{g}_{G'} ac\underline{d}b \end{array}$$

On a choisi la règle $A \to cd$. Il y a un conflit entre b et d. On fait un rebroussement et on essaye avec un autre choix, $A \to c$.

$$\begin{array}{cc} a\ c\ b & a\ c\ b \\ \triangle & \triangle \\ S \xrightarrow{g}_{G'} aAb & S \xrightarrow{g}_{G'} aAb \xrightarrow{g}_{G'} acb \end{array}$$

Cette méthode est peu efficace en cas de rebroussement (proche de la méthode naïve).

Analyseur prédictif descendant LL(k)

Prédictif : on utilise une partie du mot d'entrée pour "prédire" la règle à utiliser et de façon irrémédiable.

Analyse prédictive descendante : Analyse LL(k)
- Left-to-right scanning : on lit l'entrée de gauche à droite.
- Leftmost derivation : on construit une dérivation gauche partant de l'axiome.
- on utilise les k lettres courantes de l'entrée pour faire la prédiction.

On va principalement s'intéresser à l'analyse *LL(1)* : on ne regarde qu'une lettre de l'entrée. Il y a deux types d'analyse descendante : l'analyse *récursive* et l'analyse *non-récursive*.

Analyseur récursif LL(1)

On a une grammaire algébrique $G = (\Sigma, V, S, P)$. A toute variable $V_i \in V$, on fait correspondre une procédure $V_i()$ en charge de "reconnaître" les mots w tels que $V_i \rightarrow_G^* w$.
- les différentes procédures vont s'appeler les unes les autres selon les règles de production de la grammaire.
- Le programme d'analyse est simplement un appel à la procédure de l'axiome S.
- on place à la fin du mot à analyser un *marqueur de fin* \$.
- le pointeur $\triangle$ désignant une position dans le mot ne peut qu'avancer (pas de rebroussement) $\Rightarrow$ mot d'entrée $\simeq$ un flôt de lettres.

Plus important : selon le terminal désigné dans le mot d'entrée, pour un non-terminal donné (partie gauche de règle), il n'y a qu'une règle de production utilisable.

Pour la procédure $V_i()$:

1. selon le terminal pointé par $\triangle$ (chaque terminal est donc un cas), on considère la règle de production $V_i \rightarrow \alpha_1 \dots \alpha_n$ à utiliser
2. en séquence, pour j allant de 1 à n,
   - si $\alpha_j \in \Sigma$, alors on appelle *consommer*(α_j).
   - si $\alpha_j \in V$, on appelle la procédure $\alpha_j()$
3. si la règle à considérer est $V_i \rightarrow \epsilon$, alors on ne fait rien.

procedure *consommer*(a) :
- si $\triangle$ pointe sur le terminal a alors on avance $\triangle$
  sinon on lève exception d'erreur de syntaxe

procedure *analyse*() :
- on place le $\triangle$ sur la première lettre du mot à analyser ;
- $S()$;
- si $\triangle$ ne pointe pas sur \$ alors
  on lève une exception d'erreur de syntaxe ;

On ne peut pas construire un analyseur (récursif) LL(1) pour une grammaire G si :
- G est ambiguë
- G est récursive gauche

  $A \rightarrow Aa \mid \epsilon$: le pointeur reste sur la première lettre du mot à analyser et on ne peut choisir avec cette information assurément entre $A \rightarrow Aa$ et $A \rightarrow \epsilon$.

— G n'est pas factorisée à gauche

$A \to aA \mid aB$: si le pointeur désigne a, on ne peut choisir avec cette information assurément entre $A \to aA$ et $A \to aB$.

Récursivité à gauche

— *immédiate* : la grammaire contient un non-terminal A et une règle de production $A \to A\alpha$ ($\alpha \in (\Sigma \cup V)^+$).

— *générale* : la grammaire contient un non-terminal A et il existe une dérivation $A \to_G^+ A\alpha$ ($\alpha \in (\Sigma \cup V)^+$).

Elimination de la récursivité à gauche immédiate

On remplace les règles de la grammaire de la forme $A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$ où

— $\alpha_i \in (\Sigma \cup V)^+$ et $\beta_j \in (\Sigma \cup V)^*$

— les β_j ne commencent pas par A

par les règles

$$A \to \beta_1 A' \mid \ldots \mid \beta_m A'$$
$$A' \to \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \epsilon$$

où A' est un nouveau non-terminal.

Exemple :
$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid i$$

Après élimination de la récursivité gauche immédiate :
$$E \to TE'$$
$$E' \to +TE' \mid \epsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \epsilon$$
$$F \to (E) \mid i$$

Elimination de la récursivité à gauche générale

— On ordonne les non-terminaux : $A_1, A_2, \ldots, A_n$

— On applique l'algorithme suivant :

Pour i allant de 1 à n faire
  Pour j allant de 1 à $i-1$ faire
    on remplace la règle $A_i \to A_j\alpha$ où $A_j \to \beta_1 \mid \ldots \mid \beta_k$
    par la règle $A_i \to \beta_1\alpha \mid \ldots \mid \beta_k\alpha$.
  FinPour
  On élimine la récursivité à gauche immédiate
  pour toutes les règles de A_i.
FinPour

Premier exemple : $S \to Aa \mid b$, $A \to Ac \mid Sd \mid c$ On ordonne : S, A

— 1ère itération : $S \to Aa \mid b$, on ne change rien ...

— 2ème itération : On remplace $A \to Sd$ par $A \to Aad \mid bd$.

$$S \to Aa \mid b$$
$$A \to Ac \mid Aad \mid bd \mid c$$

Elimination de la récursivité gauche immédiate pour A :

$$S \to Aa \mid b$$
$$A \to cA' \mid bdA'$$
$$A' \to cA' \mid adA' \mid \epsilon$$

Deuxième exemple : $S \to Sa \mid TSc \mid d$
$T \to TbT \mid \epsilon$

On ordonne : S, T

— 1ère itération : $S \to Sa \mid TSc \mid d$, élimination de la récursivité immédiate pour S

$$S \to TScS' \mid dS' \qquad T \to TbT \mid \epsilon$$
$$S' \to aS' \mid \epsilon$$

— 2ème itération : $T \to TbT \mid \epsilon$, élimination de la récursivité immédiate pour T

$$S \to TScS' \mid dS' \qquad T \to T'$$
$$S' \to aS' \mid \epsilon \qquad T' \to bTT' \mid \epsilon$$

Malheureusement, $S \to_{G'} TScS' \to_{G'} T'ScS' \to_{G'} ScS'$!!

$\Rightarrow$ l'algorithme ne fonctionne pas toujours, notamment à cause des règles de la forme $X \to \epsilon$, ou plus généralement si $X \to_G^* \epsilon$. Cependant, si la grammaire est *propre*, alors cette méthode pour ôter le récursivité gauche fonctionne toujours.

Factorisation à gauche d'une grammaire

— On remplace les règles de la forme

$$X \to \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

où
— $\alpha \in (\Sigma \cup V)^+$ et $\beta_i, \gamma_j \in (\Sigma \cup V)^*$,
— le préfixe commun α est choisi le plus grand possible,
— α n'est pas préfixe des γ_j.
par les règles

$$X \to \alpha X' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$X' \to \beta_1 \mid \ldots \mid \beta_n$$

où X' est un nouveau non-terminal.

— On réitère tant que nécessaire.

Analyseur non récursif LL(1)

Dans l'analyseur récursif descendant vu précédemment, c'est la pile des appels de procédure qui est en fait utilisée pour empiler les règles (puisqu'on fait correspondre à un non terminal une procédure) et dépiler les terminaux (les procédures se terminent quand on a lu les terminaux qui leur sont associés). Pour construire un analyseur non récursif (qui sera donc un automate à pile), on construit une *table d'analyse* qui détermine le comportement de l'automate en fonction du sommet de pile et du terminal lu.

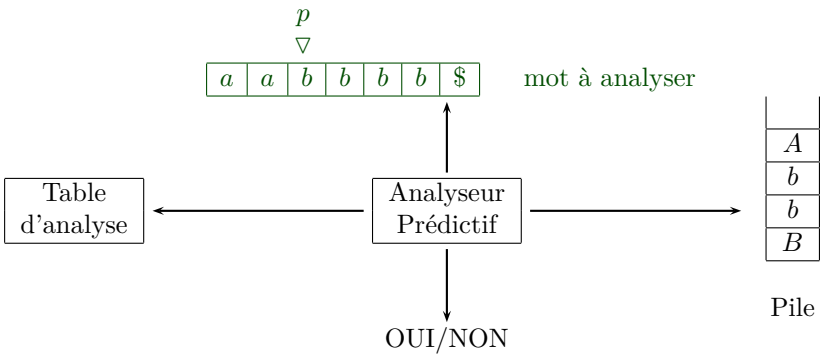


Table d'analyse

La table d'analyse contient **une entrée pour chaque couple** (a, X) ($a \in \Sigma$, $X \in V$).
L'entrée (a, X) contient **au plus** une règle de production $X \to \gamma$ qui est la règle utilisée pour dériver le non-terminal X si le pointeur p du mot à analyser pointe sur le terminal a.
Une entrée vide dans la table correspond en fait à une erreur, c'est-à-dire un mot non-engendré par la grammaire.

Fontionnement de l'analyseur non récursif

Initialement, le pointeur p désigne la première lettre du mot à analyser et la pile contient l'axiome de la grammaire.
Tant que vrai faire

- si la pile est vide alors
  - si $p \rhd \$$ alors retourner OUI
  - si $p \rhd a$ et $a \neq \$$ alors retourner NON
- si le sommet de pile est un terminal a alors
  - si $p \rhd a$ alors dépiler a et avancer p
  - si $p \rhd b$ ($b \neq a$) alors retourner NON
- si le sommet de pile est un non-terminal X alors
  - si $p \rhd a$ et l'entrée de la table d'analyse pour (a, X) est vide alors retourner NON
  - si $p \rhd a$ et l'entrée de la table d'analyse pour (a, X) contient $X \to \epsilon$ alors dépiler X
  - si $p \rhd a$ et l'entrée de la table d'analyse pour (a, X) contient $X \to \alpha_1 \dots \alpha_n$ ($\alpha_i \in (\Sigma \cup V)$) alors dépiler X ; empiler α_n ; … ; empiler α_1

FinTantque

Exemple
$G = (\{a, b\}, \{S, B\}, A, S, P)$ avec

$$P = \{S \to AB,\ A \to aAb \mid \epsilon,\ B \to bB \mid \epsilon\}$$

Table d'analyse :

	a	b	\$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \epsilon$	$A \to \epsilon$
B		$B \to bB$	$B \to \epsilon$

Construction de la table d'analyse Soit G une grammaire algébrique $G = (\Sigma, V, S, P)$. Il faut déterminer pour chaque couple $(a, S) \in \Sigma \times V$ la règle de production $S \to \gamma$ que l'automate doit utiliser. Suivant la grammaire, pour un terminal a et un non-terminal S, on peut avoir différents cas de figure :

$S \to aS \mid b$
Utiliser la règle $S \to aS$.

$S \to A \mid B$
$A \to aA \mid \epsilon$
$B \to bB \mid \epsilon$
Utiliser la règle $S \to A$.

$S \to DA \mid B$
$A \to aA \mid \epsilon$
$B \to bB \mid \epsilon$
$D \to dD \mid \epsilon$
Utiliser la règle $S \to DA$.

Le prédicat *Eps*$(w), w \in (\Sigma \cup V)^+$ (ou *Epsilon*(w))
On définit pour tout mot w de $(\Sigma \cup V)^+$ le prédicat *Eps*(w) qui est vrai ssi $w \to_G^* \epsilon$.
- *Eps*(ϵ) est vrai
- soit $\alpha_1 \dots \alpha_n$ avec $\alpha_i \in (\Sigma \cup V)$,
  - s'il existe un j tel que $\alpha_j \in \Sigma$, alors *Eps*$(\alpha_1 \dots \alpha_n)$ est faux.
  - si *Eps*$(\alpha_1 \dots \alpha_n)$ est vrai alors $\alpha_1, \dots, \alpha_n \in V_\epsilon$

En calculant V_ϵ pour la grammaire G, on peut déterminer pour un mot $w \in (\Sigma \cup V)^+$ la valeur de *Eps*(w) ($V_\epsilon = \{X \in V \mid X \to_G^* \epsilon\}$ i.e. terminaux se dérivant en ϵ).

L'ensemble *Premier*$(w), w \in (\Sigma \cup V)^+$
Pour tout mot w de $(\Sigma \cup V)^+$, on définit *Premier*(w) comme $\{a \in \Sigma \mid w \to_G^* aw'\}$.

Calcul de Premier(w) *:*
Premier(w) est le plus petit ensemble de terminaux qui vérifie :
- si $a \in \Sigma$ alors *Premier*$(a) = \{a\}$.
- si $X \in V$ et $X \to \beta \in P$ alors *Premier*$(\beta) \subseteq$ *Premier*(X).
- pour $\alpha_1 \dots \alpha_n$ (avec $\alpha_i \in (\Sigma \cup V)$), on a
  - *Premier*$(\alpha_1) \subseteq$ *Premier*$(\alpha_1 \dots \alpha_n)$.
  - Pour i allant de 1 à $n-1$ faire
    Si *Eps*$(\alpha_1 \dots \alpha_i)$ est vrai alors
    Premier$(\alpha_{i+1}) \subseteq$ *Premier*$(\alpha_1 \dots \alpha_n)$

Exemple de calcul des premiers

$S \to DA \mid B$
$A \to aA \mid \epsilon$
$B \to bB \mid \epsilon$
$D \to dD \mid \epsilon$

$Premier(D) = \{d\}, Premier(B) = \{b\}$
$Premier(A) = \{a\}$

$Premier(D) \subseteq Premier(S)$
$Premier(B) \subseteq Premier(S)$
$Premier(A) \subseteq Premier(S)$ car $D \to^*_G \epsilon$
Donc, $Premier(S) = \{a, b, d\}$.

A ce stade, la table d'analyse contiendra une règle $X \to \alpha$ pour chaque entrée $(a, X), a \in \Sigma$ telle que a appartient à $Premier(\alpha)$. En effet, l'ensemble des premiers d'un non terminal X désigne intuitivement l'ensemble des terminaux que l'on peut rencontrer comme préfixe dans une dérivation de X. On note ainsi dans la table la première règle de production utilisée dans la dérivation $X \to^*_G a\beta$ pour l'entrée (a, X).

	a	b	d	\$
S	$S \to DA$	$S \to B$	$S \to DA$	
A	$A \to aA$			
B		$B \to bB$		
D			$D \to dD$	

Cependant si cette dérivation comporte plusieurs étapes, elle utilise nécessairement des règles de production de la forme $Y \to \epsilon$. Or le calcul des premiers ne permet pas de placer les règles de production de cette forme dans la table. Il faut ici calculer les *suivants* de Y car, comme $Y \to \epsilon$, les premiers terminaux que l'on peut rencontrer en dérivant Y sont donnés par les premiers des variables apparaissant après Y dans l'ensemble des règles de production. Plus généralement ceci s'applique pour toute variable X qui peut se dériver en ϵ ($X \to^*_G \epsilon$).

L'ensemble *Suivant*$(X), X \in V$
Pour tout non-terminal X de V, on définit $Suivant(X)$ comme
$\{a \in \Sigma \mid S \to^*_G uXav\} \cup \{\$ \mid S \to^*_G uX\}$.

Calcul de $Suivant(X)$ *:*
$Suivant(X)$ est le plus petit ensemble de terminaux ou \$ qui vérifie :
- Pour l'axiome S, $\$ \in Suivant(S)$.
- si $Y \to \alpha X\beta \ \in P$ alors $Premier(\beta) \subseteq Suivant(X)$.
- si $Y \to \alpha X \ \in P$ ou $Y \to \alpha X\beta \ \in P$ avec $Eps(\beta) =$ vrai alors $Suivant(Y) \subseteq Suivant(X)$.

Exemple de calcul des suivants

$S \to DA \mid B$ $\quad Suivant(D) = Premier(A) \cup Suivant(S) = \{a, \$\}$
$A \to aA \mid \epsilon$ $\quad Suivant(B) = Suivant(S) = \{\$\}$
$B \to bB \mid \epsilon$ $\quad Suivant(A) = Suivant(S) = \{\$\}$
$D \to dD \mid \epsilon$ $\quad Suivant(S) = \{\$\}$

Ceci permet de compléter la table d'analyse précédente, en ajoutant un règle $X \to \alpha$, avec

Eps(α) =vrai, pour toute entrée (e, X) telle que e appartient à *Suivant*(X).

	a	b	d	\$
S	$S \to DA$	$S \to B$	$S \to DA$	$S \to DA$ $S \to B$
A	$A \to aA$			$A \to \epsilon$
B		$B \to bB$		$B \to \epsilon$
D	$D \to \epsilon$		$D \to dD$	$D \to \epsilon$

Construction de la table d'analyse

- Pour chaque règle de production $X \to \alpha$ ($\alpha \in (\Sigma \cup V)^+$),
  pour chaque terminal a de *Premier*(α),
  Mettre $X \to \alpha$ à l'entrée (a, X) de la table.
- Pour chaque règle de production $X \to \alpha$ avec *Eps*(α) =vrai,
  Mettre $X \to \alpha$ à l'entrée (e, X) de la table pour tout
  élément e dans *Suivant*(X) ($e \in \Sigma$ ou $e = \$$).

Grammaire LL(1) Si la table d'analyse contient au plus une règle de production par entrée, alors la grammaire est LL(1) ; dans le cas contraire, la grammaire n'est pas LL(1).

Une grammaire n'est pas LL(1) s'il existe deux règles de production $X \to \alpha$ et $X \to \beta$ telles que

- *Premier*(α) $\cap$ *Premier*(β) $\neq \varnothing$, ou
- *Premier*(β) $\cap$ *Suivant*(X) $\neq \varnothing$ avec *Eps*(α) =vrai.
- *Eps*(α) =vrai et *Eps*(β) =vrai

Attention ! ! Les conditions de non-ambiguïté, de non-récursivité gauche et de factorisation à gauche ne sont que des conditions *nécessaires* pour qu'une grammaire soit LL(1). Elles ne sont pas *suffisantes*.
La grammaire $(\{a, b\}, \{S, A\}, S, P)$ avec $P = \{S \to aSb \mid A,\ A \to aA \mid \epsilon\}$

- n'est pas ambigüe,
- n'est pas récursive gauche
- est factorisée à gauche

mais elle n'est pas LL(1).

Exercice 1. Reprendre la grammaire des expressions arithmétiques et construire la table d'analyse LL correspondante. On vérifiera ainsi que cette grammaire est bien LL(1).

$$\begin{array}{ll} E \to TE' & T' \to *FT' \mid \epsilon \\ E' \to +TE' \mid \epsilon & F \to (E) \mid i \\ T \to FT' & \end{array}$$

Exercice 2. On considère la grammaire suivante engendrant des listes *à la Lisp* :

$$\begin{array}{lcll} \mathrm{L} & \longrightarrow & \mathtt{(}\ \mathrm{S}\ \mathtt{)} & \mid\ \mathtt{a} \\ \mathrm{S} & \longrightarrow & \mathrm{L}\ \mathrm{S} & \mid\ \varepsilon \end{array}$$

Question 1. *Construire les ensembles* suivants *et* premiers *pour les variables L et S.*

Question 2. *Construire la table d'analyse LL correspondant.*