Grammaires et Analyse Syntaxique - Cours 6
Introduction à l'analyse ascendante

Ralf Treinen

treinen@irif.fr

5 mars 2025

Analyse descendante

- Nous avons vu aux cours 4 et 5 une méthode d'analyse *descendante* (angl. : *top-down*, ou dans le contexte de l'analyse grammaticale *recursive descent*) :
- L'analyse descendante construit l'arbre de dérivation en commençant par la racine, et puis ajoute dans l'arbre des enfants dans un ordre descendant et de gauche à droite.
- L'analyse descendante correspond à la construction d'une dérivation gauche.
- Cette dérivation gauche est construite dans l'ordre naturel : on commence avec l'axiome, et on termine sur le mot d'entrée.
- On a assez peu d'information pour guider la construction : Le non-terminal courant, et le symbole suivant de l'entrée.

Analyse descendante : avantages et inconvénients

- Avantage : facile à implémenter à la main quand la grammaire est LL(1).
- Inconvénient : peut nécessiter des contorsions de la grammaire quand la grammaire n'est pas LL(1).
- Des modifications de la grammaire bousculent la structure de l'arbre de dérivation.
- Un changement fondamental de la grammaire est gênant car on s'intéresse à la fin aussi à la structure trouvée par l'analyse (l'arbre de dérivation) qui maintenant ne correspond plus à la grammaire initiale.

Analyse ascendante

- L'alternative à l'analyse descendante est l'analyse *ascendante* (angl. : *bottom-up*) :
- L'analyse ascendante construit l'arbre de dérivation en commençant par les feuilles, et en faisant grandir l'arbre par combinant des arbres existants par des nouveaux nœuds.
- Ça correspond à quel type de dérivation ? Voir la suite !
- Nous avons maintenant beaucoup plus d'informations pour guider la construction de l'arbre : on a déjà des arbres de dérivations qu'il suffit de combiner.
- Pour cette raison l'analyse ascendante peut être plus puissante que l'analyse descendante.

Analyse descendante vs. analyse ascendante

- Exemple : la grammaire

$$E \rightarrow \texttt{(E+E)} \mid \texttt{(E*E)} \mid \texttt{i}$$

- Quand l'analyse *descendante* voit le symbole `(`, elle ne sait pas quelle production appliquer.
- Nous avons vu au cours 4 que cette grammaire n'est pas LL(1).
- Quand l'analyse *ascendante* voit que des parties de l'entrée sont reconnues comme étant des `(`, E, `+`, E, `)`, elle sait que ça correspond à la première production, et elle va combiner ces 5 arbres par un nouveau nœud E.

Analyse ascendante

- Elle lit l'entrée de gauche à droite.
- Le principe est que des parties de l'entrée sont combinées en des morceaux d'arbre de dérivation dès que possible.
- Structures de donnée de l'analyseur ascendant :
  - le reste de l'entrée qui reste à consommer
  - une pile de symboles de $\Sigma \cup N$. Elle représente la partie de l'entrée qu'on a déjà consommé, et pour laquelle ont a déjà construit des morceaux d'arbre.
- Toutes les réductions (applications d'une règle dans le sens ascendant) se font sur la partie haute de la pile.
- Un non-terminal sur la pile est en vérité la racine d'un morceau d'arbre de dérivation.

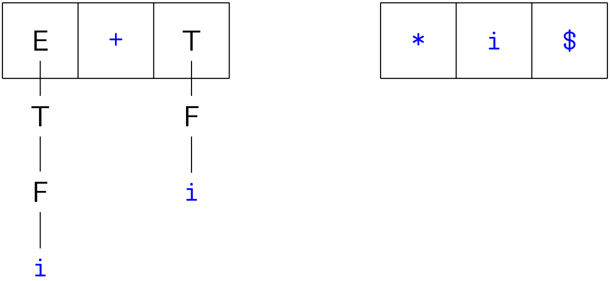
Pile et Entrée pendant l'analyse ascendante

Règles : $S \to E\ \$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (\ E\ ) \mid i$

Partie de l'entrée déjà lue : `i+i`

Reste à lire : `*i$`

pile (sommet à droite) : entrée :



Actions dans l'analyse ascendante

- Il a quatre types d'action :
  - **shift** : transférer le symbole suivant de l'entrée sur le sommet de la pile ;
  - **reduce** $N \to \alpha$: remplacer une séquence α qui se trouve en haut de la pile par un N, quand N $\to \alpha$ est une règle de la grammaire ;
  - **accepter** ;
  - signaler une **erreur**.
- On parle aussi de *shift-reduce parser* car shift et reduce sont les actions essentielles.

Shift

Situation de départ :

pile (sommet à droite) :

x_1	x_2	x_3

entrée :

a_1	a_2	a_3	a_4

Situation après un **shift** :

pile (sommet à droite) :

x_1	x_2	x_3	a_1

entrée :

a_2	a_3	a_4

Reduce

Situation de départ :

pile (sommet à droite) :

x_1	x_2	y_1	y_2	y_3

entrée :

a_1	a_2	a_3

Situation après un **reduce** N $\rightarrow y_1 y_2 y_3$:

pile (sommet à droite) :

x_1	x_2	N

entrée :

a_1	a_2	a_3

Exemple

- La grammaire $G_1 = (\{\mathtt{i}, \mathtt{+}, \mathtt{*}, \mathtt{(}, \mathtt{)}, \mathtt{\$}\}, \{\mathsf{S}, \mathsf{E}, \mathsf{T}, \mathsf{F}\}, \mathsf{S}, P)$, où P est

$$
\begin{array}{lcl}
\mathsf{S} & \rightarrow & \mathsf{E}\ \mathtt{\$} \\
\mathsf{E} & \rightarrow & \mathsf{E}\ \mathtt{+}\ \mathsf{T} \mid \mathsf{T} \\
\mathsf{T} & \rightarrow & \mathsf{T}\ \mathtt{*}\ \mathsf{F} \mid \mathsf{F} \\
\mathsf{F} & \rightarrow & \mathtt{(}\ \mathsf{E}\ \mathtt{)} \mid \mathtt{i}
\end{array}
$$

- Cette grammaire engendre les expressions arithmétiques avec $+$ et $*$, en prenant en compte la priorité de $*$ sur $+$, et l'associativité à gauche de $+$ et de $*$.
- En fait cette grammaire n'est pas LL(1), comme nous avons vu au cours 5.

Exemple : actions du parseur

- Règles de la grammaire :

$$S \to E \,\$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (E) \mid i$$

- Actions : s (shift), r (reduce), a (accept)

pile	*entrée*	*action*	*pile*	*entrée*	*action*
	i+i*i$	*s*	E+T	*i$	*s*
i	+i*i$	*r* $F \to i$	E+T*	i$	*s*
F	+i*i$	*r* $T \to F$	E+T*i	$	*r* $F \to i$
T	+i*i$	*r* $E \to T$	E+T*F	$	*r* $T \to T * F$
E	+i*i$	*s*	E+T	$	*r* $E \to E + T$
E+	i*i$	*s*	E	$	*s*
E+i	*i$	*r* $F \to i$	E$		*r* $S \to E\$$
E+F	*i$	*r* $T \to F$	S		*a*

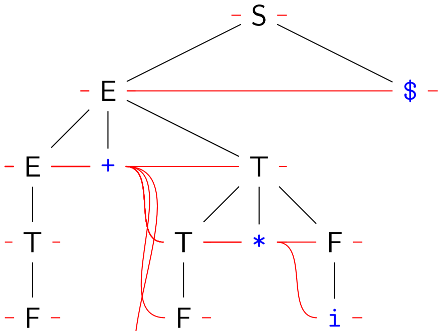
Exemple : quelle est la dérivation produite ?

- On regarde la suite des reductions sur la concaténation de pile et entrée :
  i+i*i$ – F+i*i$ – T+i*i$ – E+i*i$ – E+F*i$ – E+T*i$ – E+T*F$ – E+T $ – E$ – S
- Inverser l'ordre de cette séquence :
  S – E$ – E+T $ – E+T*F$ – E+T*i$ – E+F*i$ – E+i*i$ – T+i*i$ – F+i*i$ – i+i*i$
- C'est une dérivation droite !
- Donc, la parseur shift-reduce construit une dérivation droite dans un ordre inversé.
- Regardons cela maintenant avec construction de l'arbre de dérivation.

Exemple de l'exécution d'un parseur *shift/reduce*

S → E $ E → E + T | T T → T * F | F F → (E) | i

Situation initiale Après *Shift* Après *Reduce* F → i Après *Reduce* T → F Après *Reduce* E → T Après *Shift* Après *Shift* Après *Reduce* F → i Après *Reduce* T → F Après *Shift* Après *Shift* Après *Reduce* F → i Après *Reduce* *T* → T * F Après *Reduce* *E* → E + T Après *Shift* Après *Reduce* S → E $ *Accept !*



Efficacité

- Pour appliquer une opération de *reduce* $N \rightarrow \alpha$ il faut chercher une occurrence de α.
- L'intérêt de la construction d'une dérivation droite construite à l'envers est qu'il suffit de chercher α dans la pile.
- En plus, si on réussit à éviter des *shift* prématurés, il suffit de regarder seulement le haut de la pile !

Prendre fausse route

- Attention le parseur peut a priori aussi prendre une fausse route.
- Dans une situation où il y a en haut de la pile α et N $\rightarrow \alpha$ est une règle :
  - On peut faire un **reduce** $N \rightarrow \alpha$, ou un **shift**
  - Quand il y une autre règle $M \rightarrow \beta$, et β est également en haut de la pile (c'est possible quand α est un suffixe de β ou l'inverse) :
    on peut faire **reduce** $N \rightarrow \alpha$ ou **reduce** $M \rightarrow \beta$
- Comment éviter que notre parseur prenne fausse route ?

Shift ou Reduce ?

Configuration :

x_1	x_2	y_1	y_2	y_3		a_1	a_2

1. Soit **reduce** N $\rightarrow y_1 y_2 y_3$:

x_1	x_2	N		a_1	a_2

2. Soit **shift** :

x_1	x_2	y_1	y_2	y_3	a_1		a_2

Deux Reduce possibles

Pile :

x_1	x_2	y_1	y_2	y_3

1. Soit **reduce** N $\rightarrow y_1 y_2 y_3$:

x_1	x_2	N

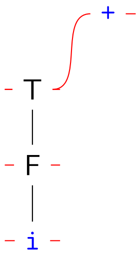
2. Soit **reduce** M $\rightarrow y_2 y_3$:

x_1	x_2	y_1	M

Exemple 1 : un shift prématuré

$$S \to E\ \$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (\ E\ ) \mid i$$

Entrée : i+i*i$



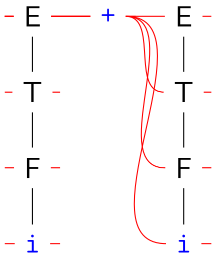
Le problème dans l'exemple 1

- Dans l'exemple 1 on a obtenu en haut de la pile : T +
- Même si ce qui suit dans l'entrée se réduit vers T on aura en haut de la pile : T + T
- On a bien S $\rightarrow$ E+T $\rightarrow$ T+T
- Mais ce n'est **pas** une dérivation droite !
- On fait on n'a **pas** que S $\xrightarrow{d}^{*}$ T+T
- On aurait du réduire le premier T à E avant de shifter le +

Exemple 2 : un reduce prématuré

$$S \to E\ \$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (\ E\ ) \mid i$$

Entrée : i+i*i$



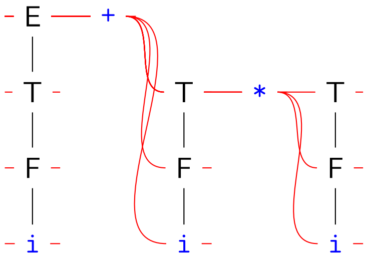
Le problème dans l'exemple 2

- Dans l'exemple 2 on a obtenu en haut de la pile : E + E
- Or, E+ se réduit seulement si on a après un T.
- Peut importe le mot $w \in \Sigma^*$ qu'on trouve après, on ne peut pas avoir que E$w \rightarrow^*$ T.
- On n'aurait pas du réduire le deuxième T à E mais shifter le $*$.

Exemple 3 : mauvais choix de la règle dans une réduction

$$S \to E\ \$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (\ E\ ) \mid i$$

Entrée : `i+i*i$`



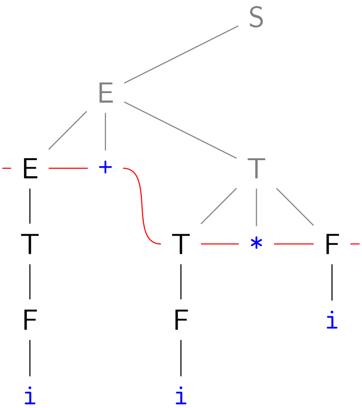
Le problème dans l'exemple 3

- Dans l'exemple 3 on a obtenu en haut de la pile : T $*$ T
- Or, T$*$ se réduit seulement si on a après un F.
- Peut importe le mot $w \in \Sigma^*$ qu'on trouve après, on ne peut pas avoir que T$w \rightarrow^*$ F.
- On n'aurait pas du réduire par la règle T $\rightarrow$ F mais réduire par la règle T $\rightarrow$ T$*$F.

La grande question qui reste

- Comment savoir, étant donnée la pile, quelle action faire quand les derniers symboles sur les pile peuvent être réduits par une production :
  - choix entre shift et reduce ;
  - dans le cas de reduce, choix de la règle quand il y en a plusieurs.
- Ça semble compliqué : il faut savoir si on arrivera à la fin de combiner tous les morceaux d'arbre en un seul arbre de dérivation.
- La surprise est : il y a une solution très efficace à ce problème due à Donald Knuth.
- Et cette solution utilise les automates du L2.

Exemple : pile E+T∗F



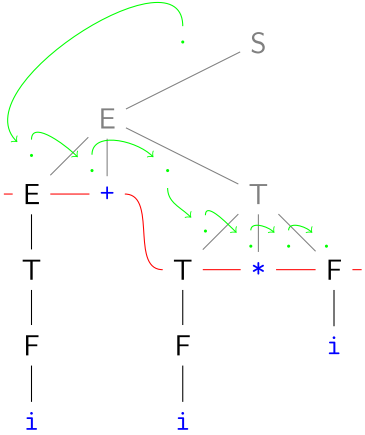
En gris la partie "à deviner".

Quelle est la structure de la partie “à deviner” ?

- C'est un morceau d'arbre qui commence par l'axiome S.
- Tous les nœuds à deviner sont étiquetés par des non terminaux, et leurs enfants constituent le côté droit d'une règle pour ce non-terminal.
- On peut dans l'arbre soit descendre vers un fils, soit passer au frère suivant.
- Quand on passe au frère suivant on vérifie que c'est justifié par un symbole qu'on trouve sur la pile.

Exemple : pile E+T∗F

$S \to E\ \$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (\ E\ ) \mid i$



Un automate

- Il s'agit de l'exécution d'un automate !
- Les états sont les points verts - l'information dans un état est la production de la grammaire, et l'endroit où on est dans le côté droit de la règle.
- L'automate peut "deviner" quelque chose car il s'agit d'un automate non déterministe.
- L'automate peut descendre par une transition ϵ, ou passer d'un enfant au suivant par une transition étiquetée par un symbole de la pile.
- On appelle cet automate l'*automate caractéristique* de la grammaire. Nous allons le définir précisément dans la suite.

Les *items*

Définition

Soit $G = (\Sigma, N, S, P)$ une grammaire. Un *item* de G est une expression de la forme

$$[N \rightarrow \alpha.\beta]$$

où $N \rightarrow \alpha\beta \in P$.
Un item de la forme $[N \rightarrow \alpha.]$ est *complet*.

- C'est à dire, un item consiste en une règle de la grammaire, plus une position (indiqué par le symbole ".") dans le côté droit de la règle.
- α et β peuvent être ϵ.
- Une règle N $\rightarrow \epsilon$ donne lieu à un seul item : $[N \rightarrow .]$

Exemple

- Soit la grammaire E $\rightarrow$ (E+E) | i   S $\rightarrow$ E \$
- Les items sont :
  [E $\rightarrow$.(E+E)], [E $\rightarrow$ (.E+E)], [E $\rightarrow$ (E.+E)],
  [E $\rightarrow$ (E+.E)], [E $\rightarrow$ (E+E.)], [E $\rightarrow$ (E+E).],
  [E $\rightarrow$ .i], [E $\rightarrow$ i.],
  [S $\rightarrow$ .E \$], [S $\rightarrow$ E.\$], [S $\rightarrow$ E \$.]
- Donc 11 états même pour une grammaire si simple.
- En général : Pour k règles de la grammaire avec longueurs des côtés droits $n_1, \ldots, n_k$, le nombre d'items est

$$\sum_{i=1\ldots k} (n_i + 1)$$

- Dans le cas de la grammaire pour les expressions arithmétiques avec priorités : 21 items.

L'automate caractéristique non-déterministe : états

- L'ensemble d'états : c'est l'ensemble des items.
- Les états initiaux sont

$$\{[\mathsf{S} \to .\alpha] \mid \mathsf{S} \to \alpha \in P\}$$

où S est l'axiome de la grammaire.

- Sur l'exemple : l'état initial est $[\mathsf{S} \to .\mathsf{E}\ \mathtt{\$}]$
- Les états acceptants sont les items complets :

$$\{[\mathsf{N} \to \alpha.] \mid \mathsf{N} \to \alpha \in P\}$$

- Sur l'exemple : les états acceptants sont :
$[\mathsf{E} \to \mathtt{(}\mathsf{E}\mathtt{+}\mathsf{E}\mathtt{)}.], [\mathsf{E} \to \mathtt{i}.], [\mathsf{S} \to \mathsf{E}\ \mathtt{\$}.]$

L'automate caractéristique non-déterministe : transitions

1. Alphabet : $\Sigma \cup N$
2. Premier type de transitions :

$$[\mathrm{N} \to \alpha.x\beta] \stackrel{x}{\Rightarrow} [\mathrm{N} \to \alpha x.\beta]$$

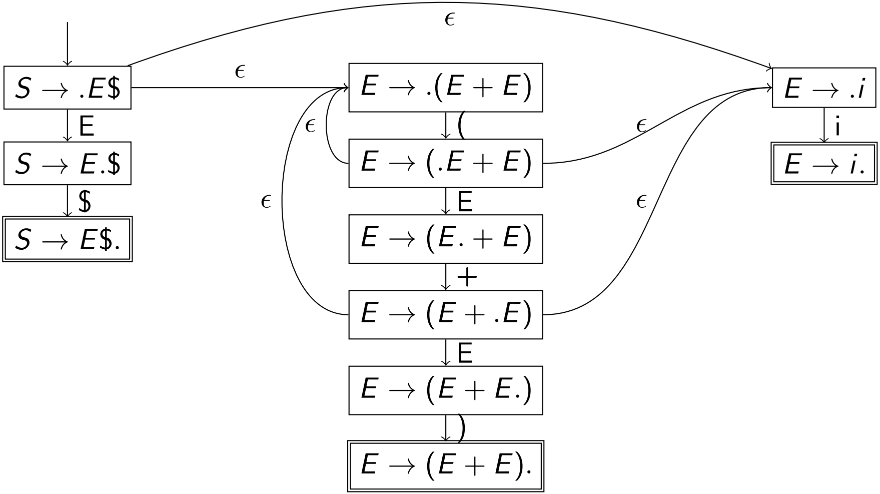
   où $\mathrm{N} \to \alpha x\beta \in P$, $x \in \Sigma \cup N$.

3. Deuxième type de transitions :

$$[\mathrm{N} \to \alpha.\mathrm{M}\beta] \stackrel{\epsilon}{\Rightarrow} [\mathrm{M} \to .\gamma]$$

   où $\mathrm{N} \to \alpha\mathrm{M}\beta, \mathrm{M} \to \gamma \in P$

L'automate caractéristique non-déterministe sur l'exemple



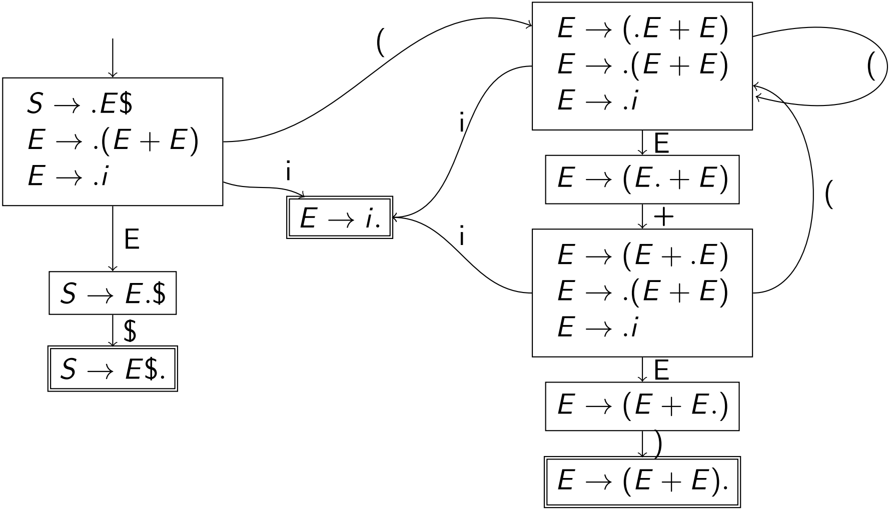
Rappel : Déterminiser et éliminer les transitions ϵ

- Soit $A = (\Sigma, Q, \delta, I, F)$ un automate non-déterministe avec ϵ-transitions.
- On définit d'abord pour $P \subseteq Q$:

$$\epsilon\text{-cloture}(P) = \{q \in Q \mid \exists p \in P, q \in \delta^*(p, \epsilon)\}$$

- On construit $A' = (\Sigma, 2^Q, \delta', I', F')$ avec
  - $I' = \epsilon\text{-cloture}(I)$
  - $\delta'(P, a) = \bigcup_{p \in P} \epsilon\text{-cloture}(\delta(p, a))$
  - $F' = \{P \subseteq Q \mid P \cap F \neq \emptyset\}$
- Il convient de construire les états de A' au fur et à mesure.

Déterminiser avec élimination des ϵ-transitions



Comment se servir de l'automate caractéristique ?

- On applique l'automate au mot sur la pile (la pile est lue du bas vers le haut).
- L'état du dernier élément de la pile peut nous dire quoi faire :
  - Si l'état contient un item complet (c.-à-d. de la forme $[N \to \alpha.]$) on peut appliquer une action **reduce** pour cette règle.
  - Si l'état contient un item non complet (c.-à-d. de la forme $[N \to \alpha.a\beta]$ et $a \in \Sigma$) on peut appliquer une action **shift**.
- C'est une solution efficace qui évite de chercher en haut de la pile : il suffit de regarder l'état au sommet de la pile.

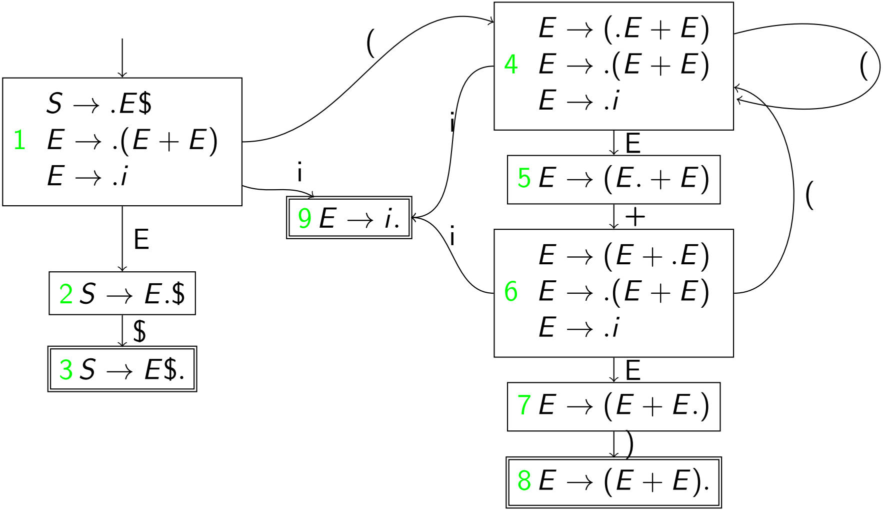
Comment se servir de l'automate caractéristique (2) ?

- Une action **reduce** remplace en haut de la pile une séquence α par un non-terminal N. Comment calculer son état ?
- On stocke sur la pile l'état avec *chaque* élément. Cela permet de calculer l'état de N, en faisant une transition de l'automate.

Définitions LR(0)

- Dans l'automate caractéristique non-déterministe on a que chaque état contient exactement un item. Après déterminisation, un état peut contenir plusieurs items !
- Un état (ensemble d'items) a un *conflit shift-reduce* quand il contient à la fois un item complet et un item incomplet où le point est devant un symbole terminal (c.-à-d. de la forme $[N \to \alpha.a\beta]$ avec $a \in \Sigma$).
- Un état (ensemble d'items) a un *conflit reduce-reduce* quand il contient deux items complets différents.
- Une grammaire est dite *LR(0)* quand son automate caractéristique déterministe n'a pas de conflit (ni shift-reduce, ni reduce-reduce).

Sur l'exemple (avec les états numérotés en vert)



C'est donc bien une grammaire LR(0) !