## 6.1 Stratégies d'analyse

Comme les automates, les grammaires définissent un langage. On peut donc, comme avec les automates, chercher pour un mot donné s'il appartient au langage engendré par une grammaire donnée.

Mais la grammaire nous donne quelque chose de plus : une décomposition, une analyse, du mot donné, en fait, un **arbre syntaxique**. Il est inutile d'expliquer à un linguiste l'intérêt de ce résultat. Il faut noter qu'alors se pose la question, non seulement de trouver une grammaire qui engendre (*i.e.* reconnaît) le langage voulu, mais aussi de trouver la grammaire qui propose la « meilleure » analyse, *i.e.* le « meilleur » arbre syntaxique. On peut en effet trouver plusieurs grammaires qui reconnaissent le même langage, ce qui les distingue alors, c'est l'arbre syntaxique associé à chaque mot. Noter que la question de la meilleure grammaire reste ouverte (plus économique ? plus « naturelle » ?...)

En tout cas, l'analyse syntaxique, étant donnés une grammaire et un mot, va donc non seulement déterminer si le mot est engendré, mais encore fournir l'arbre syntaxique correspondant.

Le problème principal est que, contrairement au automates qui peuvent être déterminisés, toute grammaire, même *context-free*, ne peut pas être « déterminisée ».

## 6.2 Analyse lexicale

Prenons le cas d'une grammaire du langage naturel (sn, sv).

La grammaire étant un dispositif puissant, mais aussi lourd (non déterministe), on préfère en général confier la tâche de la reconnaissance des symboles qui constituent un mot (nom, déterminant) à un dispositif plus simple (automate, par exemple), plutôt que d'insérer cela dans les règles.

Alors, la grammaire a pour terminaux des **tokens**, dont la décomposition est faite par un module indépendant, d'analyse lexicale.


Dorénavant, on ne s'intéressera pas beaucoup à la partie lexicale, et on fera volontiers l'hypothèse que la grammaire travaille avec des tokens comme terminaux.

Encore une remarque : l'arbre syntaxique n'est que rarement le but ultime d'une analyse : on ajoute le plus souvent une étape dite « sémantique » qui interprète le mot reconnu, en exploitant l'arbre syntaxique.

## 6.3 Analyse descendante

### 6.3.1 Algorithme

Principe : on essaie systématiquement toutes les suites de dérivation possibles à partir de $S$ (l'axiome) jusqu'à en trouver une qui correspond au mot cherché.

Il faut donc choisir une méthode de dérivation (choix de l'ordre dans lequel on applique les règles quand il y en a plusieurs possibles ; choix de l'ordre dans lequel on dérive les non terminaux).

L'algorithme implique aussi un moyen de contrôler que la dérivation en cours est compatible avec le mot cherché, et un mécanisme de *backtracking* en cas d'erreur.

**Exemple** $S \to aSbS \mid bSaS \mid \varepsilon$, mot : $abba$

Choix : d'abord le non-terminal le plus à gauche (*leftmost*). Règles considérées de gauche à droite.

Première règle :
$S \longrightarrow a\underline{S}bS \longrightarrow a\boxed{aSbS}bS$ !!! mot dérivé : $aa\ldots \neq$ mot cherché : $ab\ldots$

Deuxième règle :
$S \longrightarrow a\underline{S}bS \longrightarrow a\boxed{b\underline{S}aS}bS$ !!! mot dérivé : au moins 4 lettres, qui ne sont pas les bonnes

Troisième règle + première règle :
$S \longrightarrow a\underline{S}bS \longrightarrow a\boxed{\varepsilon}b\underline{S} \longrightarrow ab\boxed{aSbS}$ !!! mot dérivé : $aba\ldots \neq$ mot cherché : $abb\ldots$

Troisième règle + deuxième règle :
$S \longrightarrow a\underline{S}bS \longrightarrow a\boxed{\varepsilon}b\underline{S} \longrightarrow ab\boxed{bSaS}$ ... ...

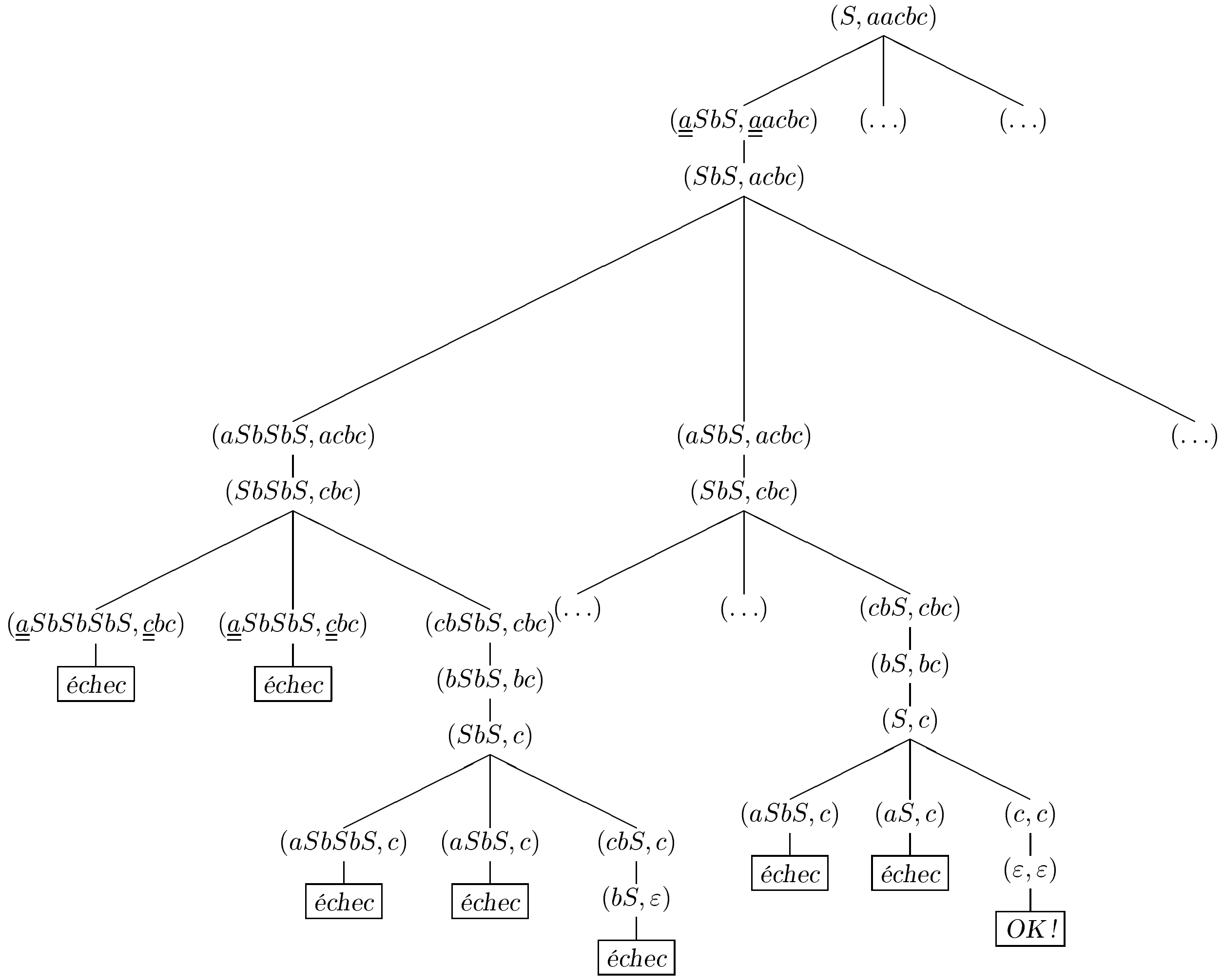
Troisième règle + deuxième règle + troisième règle + troisième règle :
$S \longrightarrow a\underline{S}bS \longrightarrow a\boxed{\varepsilon}b\underline{S} \longrightarrow ab\boxed{b\underline{S}aS} \longrightarrow abb\boxed{\varepsilon}a\underline{S} \longrightarrow abba\boxed{\varepsilon}$

**Algorithme descendant avec backtracking**

1. Pour toute production $A \to \alpha_1|\alpha_2|...|\alpha_k$, on numérote les alternatives ($\alpha_i$).
2. On utilise un pointeur sur les symboles en entrée
3. Arbre initial : axiome.
4. On utilise un pointeur sur les sommets (*sommet actif*)
   - Si le sommet actif est un non-terminal, réaliser une expansion du sommet actif en un sous-arbre dont le feuillage est $\alpha_1$. *Empiler(sommet, $i$, pointeur)*
     Nouveau sommet actif : fils gauche de $\alpha_1$.
   - Si le sommet actif est un terminal, on compare avec le symbole (d'entrée) pointé.
     - Si égalité, pointeur++, sommet actif : 1er frère droit
     - Si différence, *Depiler(sommet, $i$, pointeur)*, *Empiler(sommet, $i+1$, pointeur)*
     - S'il n'y a pas de $\alpha_{i+1}$, échec.

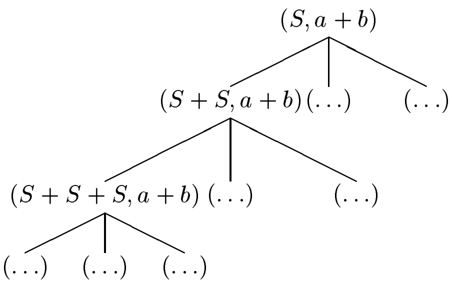
**Exemple** Grammaire $S \to aSbS \mid aS \mid c$, mot : $aacbc$

Arbre d'exploration des solutions.
Chaque nœud est un couple (*frontière de l'arbre non reconnue* ; *suffixe pas encore reconnu*).




N.B. : Si la grammaire est ambigüe, il y a plusieurs nœuds de réussite.

**Exercice** Ébaucher l'arbre d'exploration des solutions pour la grammaire $S \to S + S \mid a \mid b$ et le mot reconnu $a + b$.




Le problème qui se pose ici est le problème dit de « récursivité gauche ».