3.4 Problème du parsing

3.4.1 Algorithmes naïfs

Analyse descendante

Principe L'idée est de partir de l'axiome, et d'essayer de trouver une dérivation possible jusqu'au mot recherché.

Pour faciliter l'exploration systématique des dérivations possibles, il faut fixer une convention régissant les choix à faire : choix de l'ordre dans lequel on applique les règles ; choix de l'ordre dans lequel on dérive les non-terminaux. On choisit ici de ne considérer que les dérivations gauches (on peut envisager une version de l'algorithme avec les dérivations droites) ; il reste à ordonner les règles, de façon arbitraire.

La version la plus naïve de l'algorithme consiste donc à produire toutes les dérivations complètes (ie de S jusqu'à $u \in X^*$) (cf. section suivante).

Cependant, même dans cette version naïve, on peut essayer de faire ce que l'on appelle de la prédiction : certaines dérivations n'aboutiront pas au mot recherché, et on peut le savoir bien avant d'aller jusqu'au bout. Pour cela, il suffit de regarder les terminaux de la proto-phrase en cours de dérivation. Si la grammaire est algébrique, alors tous les symboles terminaux de la proto-phrase forment un sous-mot du mot dérivé. On peut même énoncer un résultat plus fort : toutes les suites contigües de terminaux dans la proto-phrase sont des *facteurs* du mot dérivé.

Si on fait le choix des dérivations gauches, alors on peut décider de tester systématiquement le *préfixe terminal*[1] de la proto-phrase, et de le comparer au mot recherché.

L'algorithme est clairement non déterministe (en général), et il faut donc prévoir un mécanisme de gestion des hypothèses (*backtrack*).

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 : $abb\ldots$

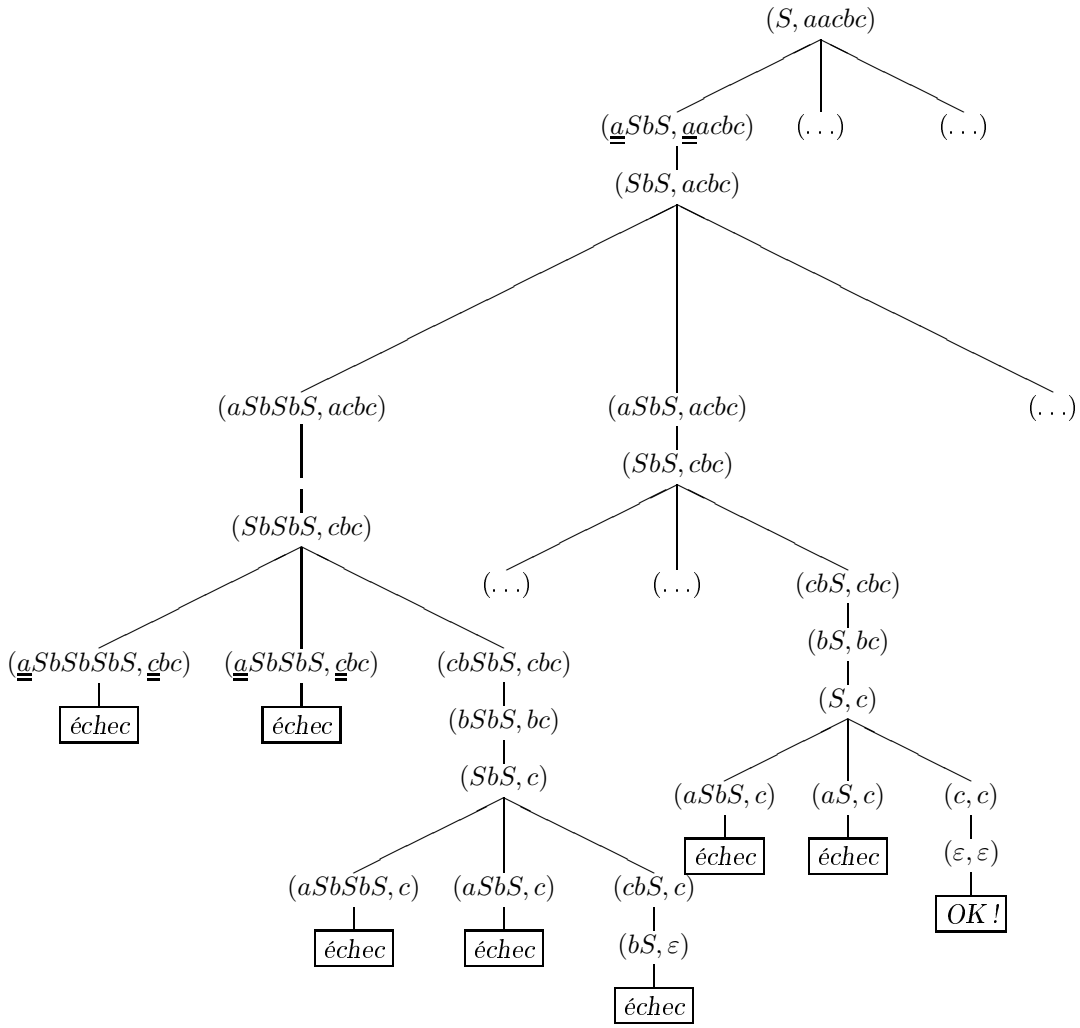
Troisième règle + deuxième règle :
$S \longrightarrow a\underline{S}bS \longrightarrow a\boxed{\varepsilon}b\underline{S} \longrightarrow ab\boxed{bSaS}$ ... ...

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

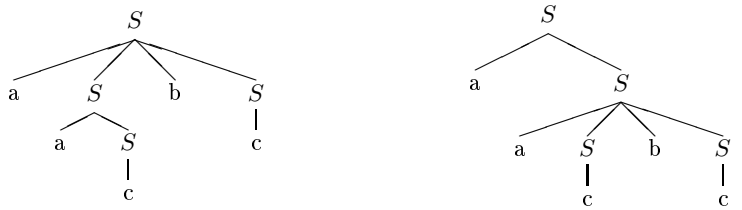
[1] Il s'agit du plus long facteur gauche f de la proto-phrase tel que $f \in X^*$.

L'algorithme avec backtracking peut être vu comme le parcours d'un arbre, l'« arbre d'exploration des solutions », dont les nœuds sont formés de couples (u, v), où u est la proto-phrase dérivée, et v le mot recherché. Au départ, u est l'axiome, et l'algorithme réussit si $u = v$. Pour simplifier la lecture, on peut se contenter de noter à chaque étape les suffixes (de la proto-phrase et du mot) concernés.

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



N.B. : Si la grammaire est ambigüe, il y a plusieurs nœuds de réussite. C'est le cas ici, où on a les deux arbres suivants (au moins) :



En exercice, ou en exemple, on ébauchera 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 lié à la **récursivité gauche** de la grammaire.

Algorithme Voici une version de l'algorithme qui utilise la récursivité pour parcourir l'arbre d'exploration des solutions. On s'arrête ici à la première solution trouvée. Il faut ajouter des ordres d'écriture pour donner l'arbre syntaxique (ou la dérivation).

```
parse_td(u, v)
{
    si (u = v) return true
    si (u = ε ou v = ε) return false
    si (u = Aα) {
        pour toute règle A → β,
          et tant que x := parse_td(βα, v) ≠ true
              ;
        return x ;
    }
    sinon (u = wAα) (avec w ∈ X+)
        si (v = wβ) return parse_td(Aα, β)
        sinon return false
}
```

Version sans prédiction, où l'on ne s'arrête que lorsque la proto-phrase n'est plus dérivable.

```
// la fonction principale est tdlrp : top-down left-right parsing
// α est la protophrase courante, u l'entrée à reconnaître
tdlrp(α, u)
begin
    if (α = u) then return true
    α = u1u2 ... ukAγ
    while (∃A → β) do
        if tdlrp(u1u2 ... ukβγ)=true then return true
    return false
end.
```

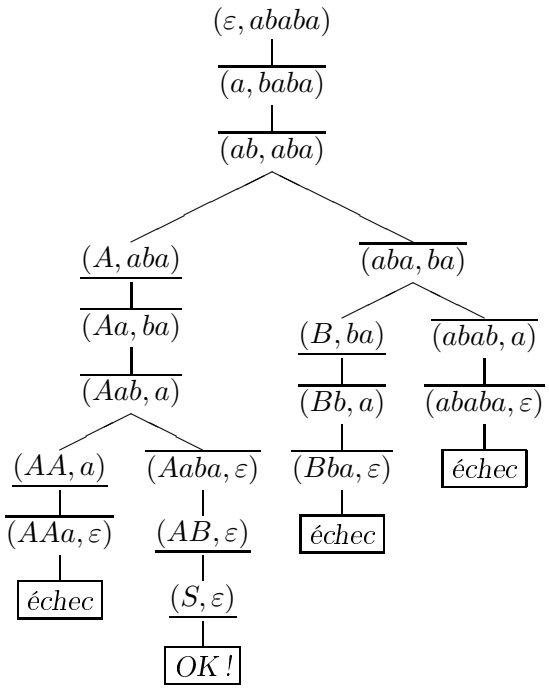
Analyse ascendante

L'idée est cette fois de "partir du bas" (analyse « *bottom-up* ») : étant donnés les premiers symboles du mot, on peut trouver une certain nombre de règles qui sont susceptibles de les avoir produits : toutes les règles dont la partie droite commence par ces symboles.

La méthode utilisée est basée sur deux « opérations » appelées **transfert** et **réduction** (*shift/reduce*).

On peut voir l'algorithme à l'œuvre sur un exemple, où est représenté l'arbre d'exploration des solutions. Chaque nœud est formé d'une paire (u, v), où $u \in (X \cup V)^*$ est la proto-phrase en cours de construction, et $v \in X^*$ est le suffixe non encore analysé du mot. À chaque étape, on a le choix (théorique) entre une **réduction** (s'il existe une règle $A \to u$, on remplace u par A (il peut y avoir plusieurs possibilités)) et un **transfert** (on ajoute le symbole le plus à gauche de v à la proto-phrase u).

Soit la grammaire $S \rightarrow AB$; $A \rightarrow ab$; $B \rightarrow aba$, et le mot à analyser *ababa*. Les nœuds résultant d'un transfert sont surlignés, et soulignés s'ils viennent d'une réduction.



En exercice, on pourra écrire l'algorithme de parcours de cet arbre. Attention, les sources d'indétermination sont nombreuses : non seulement on a le choix entre réduction et transfert (ce dernier étant toujours possible), mais il peut y avoir plusieurs réductions possibles, soit parce que plusieurs règles donnent la même partie droite, soit parce que plusieurs suffixes de la proto-phrase peuvent être réduits. (Par exemple, avec l'arbre précédent, on pourrait avoir les deux règles $X \rightarrow Aaba$ et $Y \rightarrow aba$, qui donnent deux possibilités de réduction de la proto-phrase $Aaba$: soit X, soit AY.)

Les grammaires récursives gauches ne posent pas de difficulté pour ces algorithmes. En revanche, si la grammaire n'est pas propre, et en particulier si elle contient (a) des productions singulières, ou (b) des ε-productions, on peut avoir une augmentation de la complexité, voire une non-terminaison de l'algorithme.

3.4.2 Analyse prédictive

L'idée fondamentale de ces algorithmes est de réduire l'espace de recherche, en ne descendant pas dans les branches non productives de l'arbre. Il s'agit donc d'être capable d'anticiper les conséquences d'une dérivation (analyse descendante) ou d'une réduction (analyse ascendante). On se base pour cela, dans le cas général, sur des tables prédictives construites à l'avance.

Dans ce cas, alors que le parcours brutal de l'arbre d'exploration est exponentiel (par rapport à la longueur du mot), que les versions les moins naïves des algorithmes plus haut peuvent être polynomiales, on atteindra ici, dans les cas les plus favorables, des algorithmes d'analyse linéaires.

Analyse LL(k)

Le non-déterminisme dans les analyses descendantes vient du fait que, étant donné un non-terminal A à réduire, on a le choix entre toutes les parties droites des règles $A \to \beta$. On a vu que si β commence par un terminal, une décision sur la productivité de la dérivation peut être prise immédiatement (en comparant ce terminal à la chaîne à reconnaître). Mais si β commence par un non-terminal, on ne peut prendre aucune décision.

Quelle(s) forme(s) peuvent avoir les grammaires pour permettre une analyse descendante (presque) déterministe ?

grammaires SLL(1) Grammaires dont toutes les productions sont de la forme $A \to a\alpha$ (avec $a \in X$, et $\alpha \in (X \cup V)^*$) ; et telles que s'il existe deux dérivations $A \to a_1\alpha_1$ et $A \to a_2\alpha_2$, alors $a_1 \neq a_2$.

Dans ce cas, il suffit de regarder un symbole (*lookahead-1*) en avant pour décider quelle dérivation s'applique. L'analyse descendante devient linéaire. Problème : toutes les grammaires algébriques n'ont pas une grammaire SSL(1) équivalente.

grammaires en forme normale de Greibach Grammaires dont toutes les productions sont de la forme $A \to a\alpha$ (avec $a \in X$, et $\alpha \in (X \cup V)^*$).

Cette fois, toute grammaire algébrique peut être « normalisée ». L'analyse est polynomiale, mais l'arbre syntaxique est très différent de celui de la grammaire initiale.

grammaires LL(k) L'idée est de construire, à partir de la grammaire initiale (sans la modifier), une **table de prédiction** : étant donné le non-terminal à réduire, et le symbole courant du mot à reconnaître, cette table donne toutes les dérivations possibles.

S'il n'y a qu'une dérivation possible dans chaque cas (au plus), la grammaire est dite LL(1). Si on peut construire une table où il n'y a qu'une dérivation possible en regardant 2 caractères, la grammaire est dite LL(2).

Les grammaires LL(k) induisent une hiérarchie stricte de langages.

Exemple de table de prédiction :

Grammaire $S \to aSb$; $S \to cC$; $C \to dC$; $C \to c$.

	a	b	c	d
S	aSb		cC	
C			c	dC

Analyse LR(k)

Il y a plusieurs sources de non déterminisme dans l'analyse ascendante, mais on peut les résumer ainsi : il faut savoir quand réduire, et avec quelle règle.

Plus précisément, une situation idéale serait que l'on puisse connaître pour chaque production $A \to \beta$ la liste des proto-phrases qui peuvent être réduites de façon valide à A.

Exemple : grammaire
$$\begin{array}{lcl} S & \to & AB \\ A & \to & aA \mid b \\ B & \to & bB \mid a \end{array}$$

Ici, on voit facilement que le seul cas où la règle $S \rightarrow AB$ peut donner lieu à une réduction est quand la proto-phrase est AB. Mais pour $A \rightarrow aA$, il y a plusieurs cas : toutes les piles de la forme $aa \ldots aA$ peuvent être réduites en A (de façon productive). Pour la production $B \rightarrow bB$, on voit que la pile doit nécessairement être de la forme $Ab \ldots bB$ pour que la réduction $bB \rightsquigarrow B$ soit licite.

On montre que les piles valides pour une réduction peuvent toujours être décrites par une expression rationnelle, ce qui conduit à décrire cette information sous la forme d'un automate. C'est la table de transition de cet automate qui constituera la table de prédiction.

De la même façon que plus haut, on définira la classe LR(0), qui est telle que la table de prédiction comprend au plus une dérivation. Une grammaire LR(0 peut être analysée de façon déterministe.

Une grammaire LR(1) est une grammaire dans laquelle il y a des conflits (shift/reduce ou reduce/reduce) dans la table qui peuvent être levés en regardant un symbole en avant.

On peut définir des grammaires LR(2), mais on montre que le pouvoir expressif des grammaires LR(1) correspond à tous les langages algébriques non intrinsèquement ambigus.

3.4.3 Analyse du LN

Le langage naturel a le mauvais goût d'être ambigu : alors toute grammaire algébrique qui le représente est ambiguë, et les stratégies déterministes se trouvent inadaptées.

Cette ambiguïté masssive, ainsi que l'intérêt pour des analyses partielles dans le cas où le mot n'est pas dans le langage, conduisent à s'intéresser à des méthodes de parsing dites **tabulaires**, basées sur une représentation (en table) des sous-chaînes bien formées du mot à analyser.

Exemple :

4	S		V, GN
3	GN	N	
2	Det		
	ma	sœur	mange
	1	2	3

S → GN GV
GN → Det N
GV → V
Det → ma
N → sœur
V → mange

CYK : Cocke, Younger et Kasami (1967) : parse avec une grammaire en forme normale de Chomsky. Algorithme ascendant, $0(n^3)$.

Earley (1970) : ascendant, $0(n^3)$, plus approprié pour la gestion des erreurs.

Nombreuses variantes des méthodes tabulaires développées aujourd'hui ; alternative aux méthodes tabulaires : forêts d'analyses (LR généralisé), dues à Tomita 1986.