Compilation

IV. Analyse syntaxique descendante

Jacques Farré

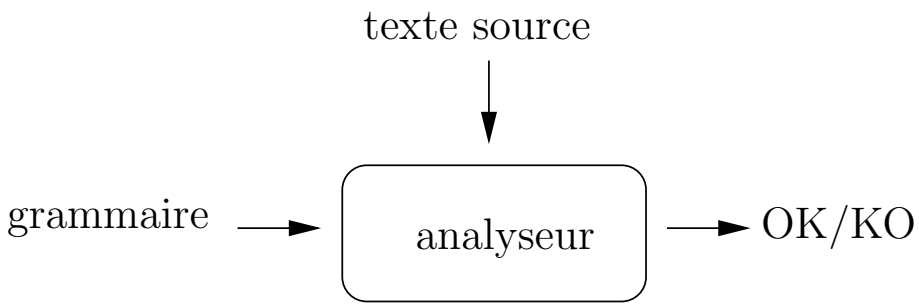
Jacques.Farre@unice.fr

http://deptinfo.unice.fr/~jf/Compil-L3

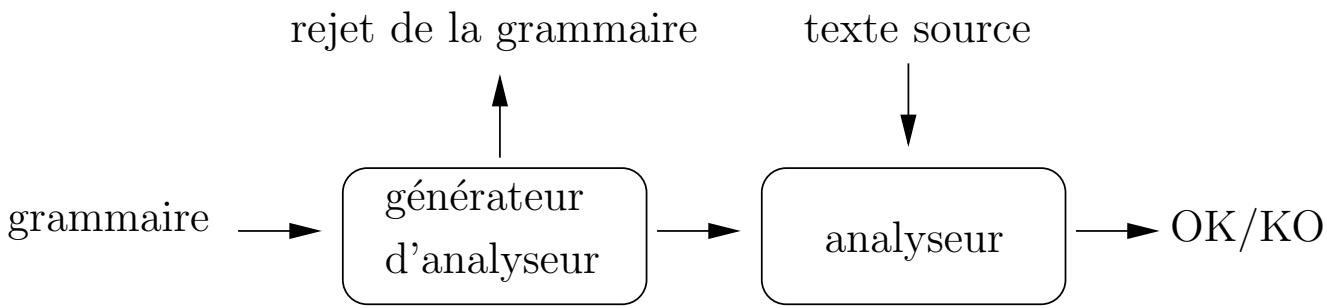
Introduction

- l'analyse syntaxique a pour but
  - de vérifier que le texte d'entrée est conforme à la grammaire
  - d'indiquer les erreurs de syntaxe et éventuellement de poursuivre l'analyse après une erreur (*reprise sur erreur*)
  - de construire une représentation intermédiaire pour les autres modules du compilateur
- elle ne s'occupe pas des aspects contextuels de la syntaxe, par exemple détecter l'erreur dans `int i ; ... ; i = "abcd" ;`
  - c'est le rôle de la *sémantique statique*, qui travaille sur la représentation intermédiaire
- afin d'avoir une représentation unique du texte source, l'analyse doit être déterministe, si possible linéaire (et éviter notamment les retours en arrière)

Génération d'analyseurs syntaxiques



« *interprétation* » *de la grammaire (CYK, Earley, . . .)*



« *compilation* » *de la grammaire*

problème : alors qu'une phrase du texte d'entrée est finie, l'ensemble des phrases d'un langage est en général infini et les phrases de longueur non bornée

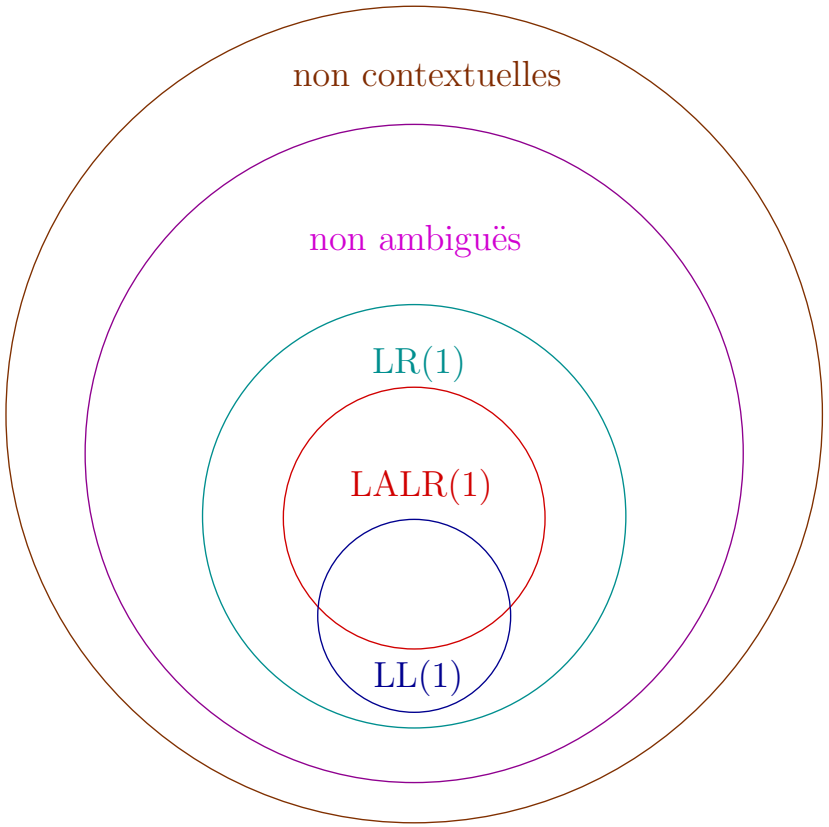
Rappel de quelques notations

- la fin du texte (par ex. la fin du fichier) sera notée par $\dashv$
- la chaîne vide sera notée ε
- l'ensemble vide sera noté $\emptyset$
- dans la suite, nous travaillons avec la grammaire *augmentée* $G' = (N', T', P', S')$ de $G = (N, T, P, S)$ avec $T' = T \cup \{\dashv\}$ $N' = N \cup \{S'\}$ et $P' = P \cup \{S' \rightarrow S \dashv\}$
- si ω est une chaîne de V^*, $k : \omega$ notera le préfixe de longueur k de ω (ou ω lui-même si $|\omega| < k$)
- une dérivation gauche sera notée $\underset{\text{lm}}{\Rightarrow}$

Types d'analyseurs déterministes

- 2 méthodes pour obtenir un analyseur syntaxique *descendant*, ou analyseur *prédictif*
  - par *descente récursive*
    - soit écriture à la main de l'analyseur
    - soit utilisation d'un générateur d'analyseur, par exemple *JavaCC*, *ANTLR*, *LLGen* ...
  - par un automate à pile explicite
  - acceptent les grammaires dites *LL(k)* ($k = 1$ en général)
- pratiquement, une seule méthode pour obtenir un analyseur *ascendant* ou par *décalage-réduction* : *LR* avec ses variantes (SLR, LALR ...)
  - représentants : *YACC/Bison*
  - accepte les grammaires *LR(k)* ($k = 1$), en fait plutôt *LALR(1)*
- il existe d'autres méthodes plus rares ou plus spécialisées : *parsing expression grammars* (impliquant des retours en arrière), méthodes propres au traitement des langues naturelles (méthodes tabulées, à charte, grammaires probabilistes ...)

Hiérarchie des grammaires



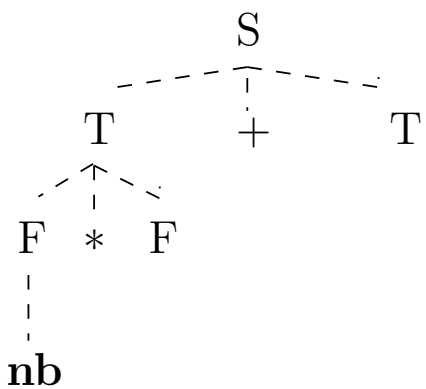
LL : lecture du texte de gauche à droite selon des dérivations gauches

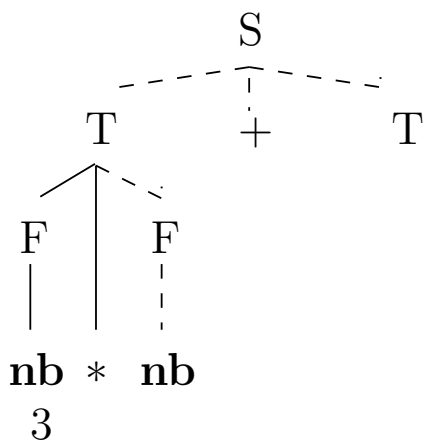
LR : lecture du texte de gauche à droite selon des dérivations droites

tout langage déterministe (c'est-à-dire reconnaissable par un automate à pile déterministe) a une grammaire LR(1)

Analyse descendante

- l'analyse descendante part de l'axiome et tente de reconstituer (virtuellement) l'arbre de dérivation par un parcours gauche-droite préfixé
  - ou, en d'autres termes, elle essaye de dériver l'axiome par une suite de dérivations gauches
- par exemple avec $\{S \rightarrow T + T,\ T \rightarrow F * F,\ F = \mathbf{nb}\}$ et la phrase 3*4+5*7

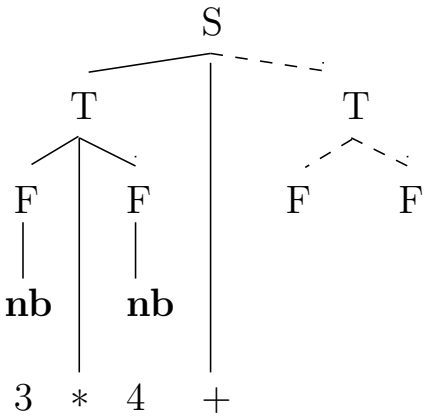




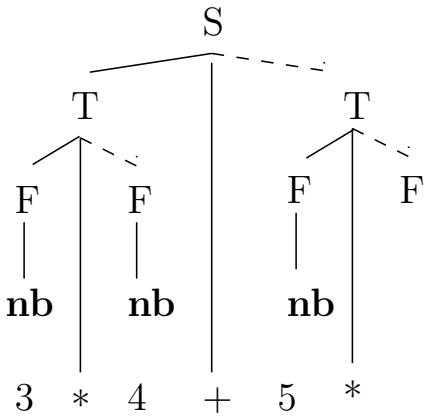
$S \underset{\text{lm}}{\Rightarrow} T + T \underset{\text{lm}}{\Rightarrow} F * F + T \underset{\text{lm}}{\Rightarrow} nb * F + T$

$\underset{\text{lm}}{\Rightarrow} nb * nb + T$

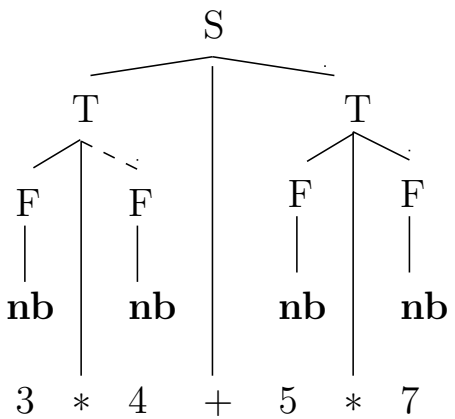
Analyse descendante – suite



$nb * nb + T \underset{\text{lm}}{\Rightarrow} nb * nb + F * F$



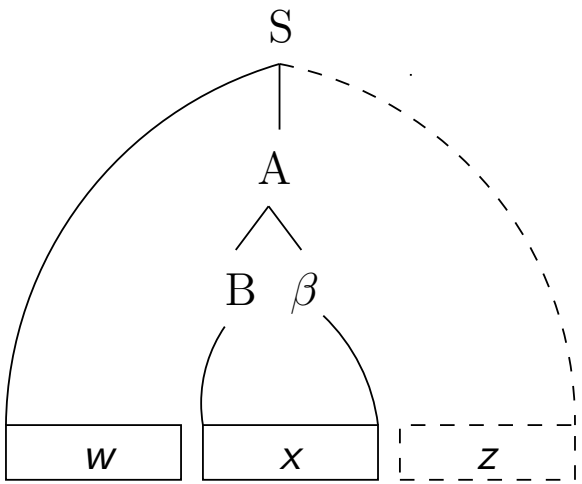
$\underset{\text{lm}}{\Rightarrow} nb * nb + nb * F$

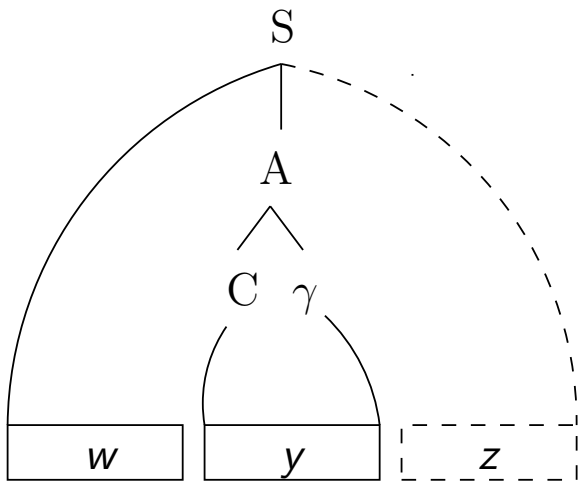


$S \underset{\text{lm}}{\Rightarrow} T + T \underset{\text{lm}}{\Rightarrow} F * F + T \underset{\text{lm}}{\Rightarrow} nb * F + T \underset{\text{lm}}{\Rightarrow} nb * nb + T \underset{\text{lm}}{\Rightarrow} nb * nb + F * F \underset{\text{lm}}{\Rightarrow} nb * nb + nb * F \underset{\text{lm}}{\Rightarrow} nb * nb + nb * nb$

Problèmes avec l'analyse descendante

- problèmes pour un analyseur descendant déterministe
  - si un non terminal a plusieurs règles, laquelle choisir ? exemple avec $A \rightarrow B\beta \mid C\gamma$: essayer $B\beta$ ou $C\gamma$?

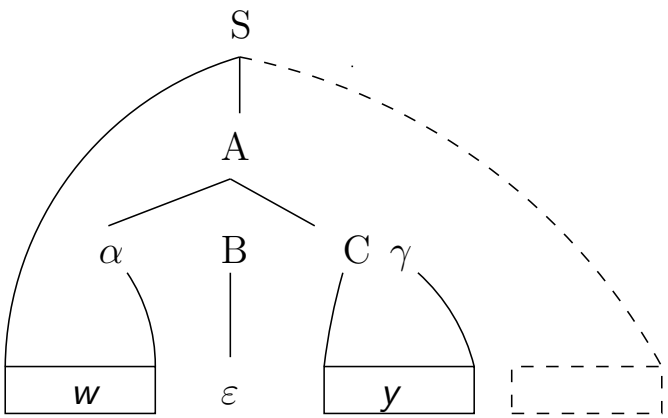


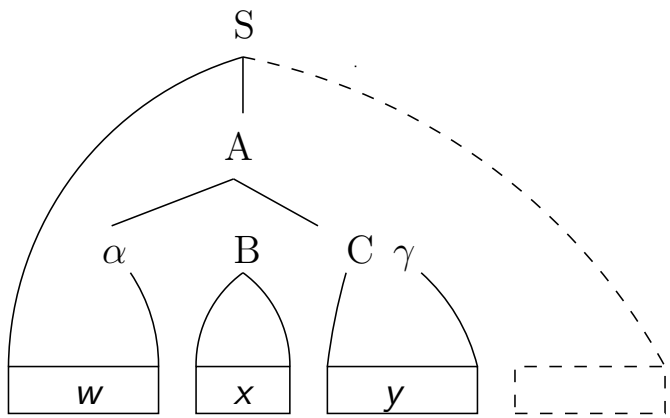


  - pour *prévoir* le choix, il faut $x \neq y$, pour **tout** x que peut produire $B\beta$ et **tout** y que peut produire $C\gamma$
  - mais il peut y avoir un nombre infini de x et de y !

Problèmes avec l'analyse descendante – suite 1

- problèmes pour un analyseur descendant déterministe
  - si un non terminal est annulable, faut-il considérer qu'il produit le vide ou non ? exemple, α ayant été reconnu : $A \rightarrow \alpha B C \gamma, B \rightarrow \varepsilon \mid X\sigma$: essayer C ou $X\sigma$?

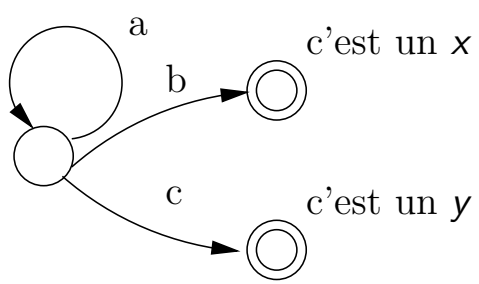




- ici encore, tous les x doivent être différents de tous les y.

Problèmes avec l'analyse descendante – suite 2

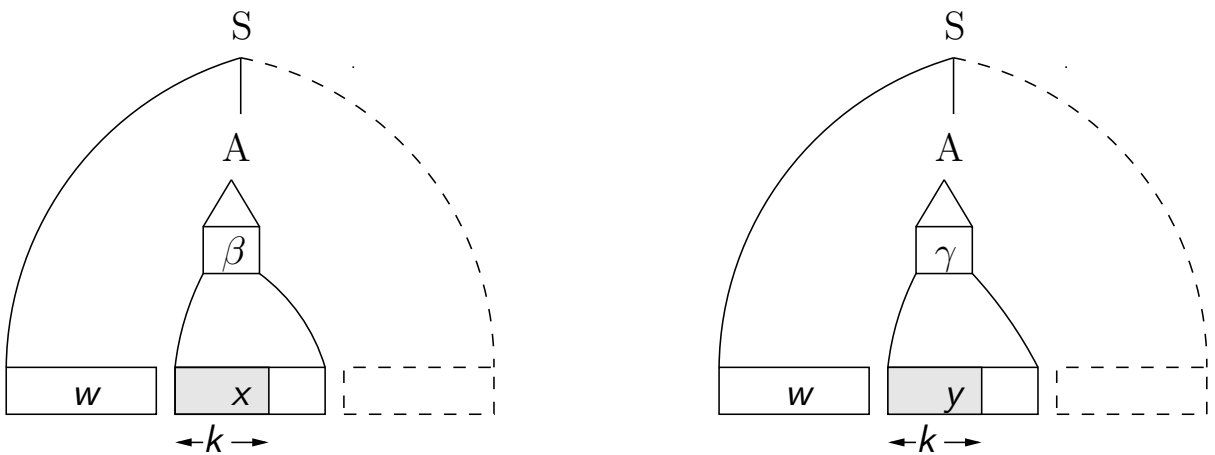
- dans le cas général, on ne peut pas vérifier $x \neq y$, $\forall x, y$
  - par exemple, les x sont de la forme a^*b et les y de la forme a^*c il faudrait vérifier que $a^k b \neq a^k c$ pour $k = 0, 1, 2, ...$
- pour avoir un choix entre un *nombre fini* de cas, soit
  - solution simple : on se limite à regarder les k premières lettres des x et des y (en d'autres termes, les k-préfixes $k : x$ et $k : y$) : k étant donné et le vocabulaire étant fini, on a à comparer des ensembles finis
  - on essaye de construire un automate fini reconnaissant les x et les y et capable de faire la différence



mais le langage des x ou des y n'est peut-être pas rationnel !

Grammaires LL(k)

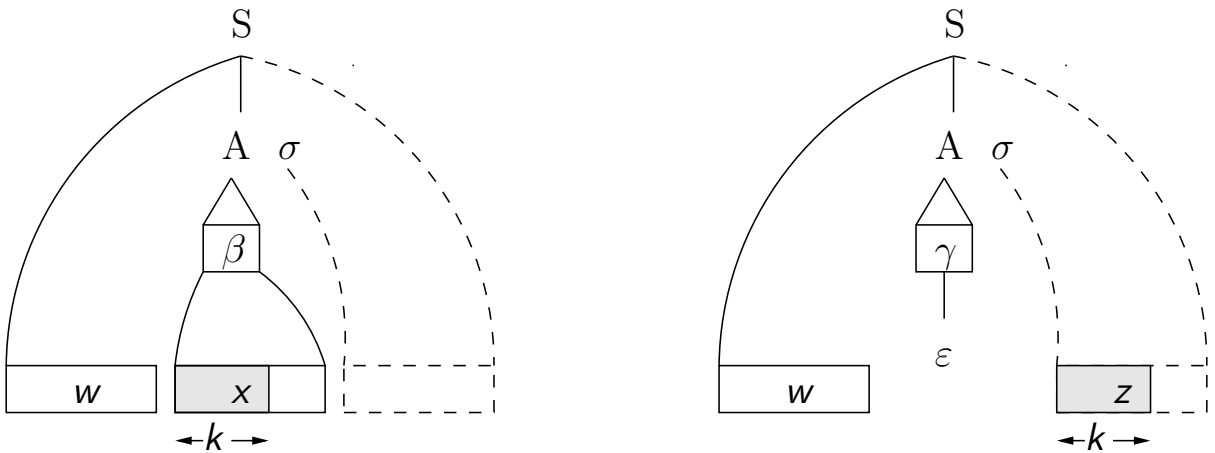
- l'analyseur regarde les k prochains symboles pour décider quelle règle appliquer
  - pour le choix $A \rightarrow \beta \mid \gamma$, il faut que β produise des mots tous différents dans leurs k premières lettres de ceux produits par γ :



  - autrement dit $\forall x, y$ tels que $\beta \Rightarrow^* x, \gamma \Rightarrow^* y, k : x \neq k : y$

Grammaires LL(k) – suite 1

- de plus, si par exemple γ est annulable, il faut que les k premières lettres de tout ce qui peut suivre A soient différentes de celles des mots produits par β



- autrement dit $\forall x, z, k : x \neq k : z$

Grammaires LL(k) – suite 2

- donc, pour les dérivations suivantes et un $k \geq 0$,

$$S \underset{\text{lm}}{\Rightarrow}^* wA\sigma \underset{\text{lm}}{\Rightarrow} w\beta\sigma \underset{\text{lm}}{\Rightarrow}^* wx$$

$$S \underset{\text{lm}}{\Rightarrow}^* wA\sigma \underset{\text{lm}}{\Rightarrow} w\gamma\sigma \underset{\text{lm}}{\Rightarrow}^* wy$$

  la grammaire est *LL(k)* si $k : x = k : y$ implique $\beta = \gamma$

  ou autrement dit, si $\beta \neq \gamma$, alors il n'existe pas x et y tels que $k : x = k : y$

- on en déduit $\text{LL}(k) \subset \text{LL}(k+1)$
  - si $k : x \neq k : y$, alors $k+1 : x \neq k+1 : y$
  - mais la réciproque n'est pas vraie

Exemple d'une grammaire LL(2) mais pas LL(1)

- soit $\mathcal{G}_1$: $S \longrightarrow aAa \mid cAba, A \longrightarrow b \mid \varepsilon$
  - il est clair que le b ne permet pas de choisir entre $A \longrightarrow b$ ou $A \longrightarrow \varepsilon$: la grammaire n'est pas LL(1)

$$S \Rightarrow c\,A\,ba \Rightarrow c\,ba \qquad S \Rightarrow c\,A\,ba \Rightarrow c\,b\,ba$$

  - par contre si b est suivi de b, c'est $A \longrightarrow b$ qu'il faut choisir, s'il est suivi de a, c'est selon la première lettre du mot :

$$\begin{array}{lll} S \Rightarrow aAa \Rightarrow aa & S \Rightarrow aAa \Rightarrow aba & b \neq a \\ S \Rightarrow cAba \Rightarrow cba & S \Rightarrow cAba \Rightarrow cbba & ba \neq bb \end{array}$$

- les grammaires ayant une récursivité à gauche ne peuvent être LL(k) pour tout k fini : $S \underset{\text{lm}}{\Rightarrow} Sa \underset{\text{lm}}{\Rightarrow} Saa \underset{\text{lm}}{\Rightarrow} \cdots \underset{\text{lm}}{\Rightarrow} baa \cdots aa$
  le choix entre $S \longrightarrow Sa$ et $S \longrightarrow b$ ne peut être déterminé que par le nombre de a, qui est non borné
- que peut-on dire des grammaires LL(0) ?

Grammaires *fortement* LL(k)

- on voit que pour une grammaire LL(k), ce qu'on a déjà lu importe pour savoir quels symboles permettront le choix
- on peut s'affranchir de cette contrainte en modifiant légèrement les conditions :
  pour les dérivations suivantes et un $k \geq 0$,

$$S \underset{\text{lm}}{\Rightarrow}{}^* wA\sigma \underset{\text{lm}}{\Rightarrow} w\beta\sigma \underset{\text{lm}}{\Rightarrow}{}^* wx$$

$$S \underset{\text{lm}}{\Rightarrow}{}^* w'A\sigma' \underset{\text{lm}}{\Rightarrow} w'\gamma\sigma' \underset{\text{lm}}{\Rightarrow}{}^* w'y$$

  la grammaire est *fortement LL(k)* si $k : x = k : y$ implique $\beta = \gamma$
  - on s'intéresse donc aux mots produits par β et γ et à ceux qui peuvent suivre A indépendamment du contexte de A
- la grammaire précédente $\mathcal{G}_1$ n'est pas fortement LL(2)
  - si on oublie le contexte gauche de A (ici a ou c), ba peut prédire $A \rightarrow b$ ($S \Rightarrow aAa \Rightarrow aba$) et $A \rightarrow \varepsilon$ ($S \Rightarrow cAba \Rightarrow cba$)

Les notions de *premiers* et de *suivants*

- les définitions précédentes impliquent 2 notions :
  - $Prem_k$: les k premières lettres qui peuvent être produites par une chaîne :
    $Prem_k(\alpha) = \{x \mid \alpha \Rightarrow^* z, x = k : z\}$
  - $Suiv_k$: les k lettres qui peuvent suivre une chaîne :
    $Suiv_k(\alpha) = \{x \mid S \Rightarrow^* w\alpha z, x = k : z\}$
- en pratique, on se contente de $k = 1$, et dans la suite, on ne mentionnera pas k

  le symbole à examiner est appelé *symbole de fenêtre* (*lookahead*) ou plus simplement, *la fenêtre*

Calcul des symboles annulables

marquer tous les symboles de V comme non annulables
répéter
  pour chaque $A \rightarrow X_1 \cdots X_n$ $(n \geq 0)$ et A non marqué annulable
    nul := vrai
    pour $i := 1 \cdots n$ **si** X_i non annulable **alors** nul := faux
    si nul = vrai **alors** marquer A comme annulable
tantque un nouveau symbole a été marqué annulable

NB. on peut accélérer l'algorithme en n'examinant pas les règles qui ont un $X_i \in T$ puisqu'un terminal n'est pas annulable

exemple $(\mathcal{G}_2)$: $S \rightarrow BABAC, A \rightarrow a, B \rightarrow b \mid DC, C \rightarrow c \mid E, D \rightarrow d \mid \varepsilon, E \rightarrow e \mid \varepsilon$

1. D et E marqués annulables car $D \rightarrow \varepsilon$ et $E \rightarrow \varepsilon$
2. C marqué annulable car $C \rightarrow E$
3. enfin, B marqué comme annulable car $B \rightarrow DC$

Calcul des *premiers*

pour $a \in T$ $Prem(a) := \{a\}$; **pour** $A \in N$ $Prem(A) := \emptyset$
répéter
  pour chaque $A \rightarrow X_1 \cdots X_n$ $(n \geq 0)$
    vide := vrai
    pour $i = 1 \cdots n$ **et tant que** vide = vrai
      ajouter $Prem(X_i) - \{\varepsilon\}$ à $Prem(A)$
      si $\varepsilon \notin Prem(X_i)$ **alors** vide := faux
    si vide = vrai **alors** ajouter ε à $Prem(A)$
tant que un $Prem(A)$ a été modifié

exemple $(\mathcal{G}_2)$: $S \rightarrow BABAC, A \rightarrow a, B \rightarrow b \mid DC, C \rightarrow c \mid E, D \rightarrow d \mid \varepsilon, E \rightarrow e \mid \varepsilon$

$Prem(S)$	$Prem(A)$	$Prem(B)$	$Prem(C)$	$Prem(D)$	$Prem(E)$
$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	$\{d, \varepsilon\}$	$\{e, \varepsilon\}$
$\{b\}$	$\{a\}$	$\{b, d, c\}$	$\{c, e, \varepsilon\}$	$\{d, \varepsilon\}$	$\{e, \varepsilon\}$
$\{b, c, d\}$	$\{a\}$	$\{b, c, d, e, \varepsilon\}$	$\{c, e, \varepsilon\}$	$\{d, \varepsilon\}$	$\{e, \varepsilon\}$
$\{b, c, d, e, a\}$	$\{a\}$	$\{b, c, d, e, \varepsilon\}$	$\{c, e, \varepsilon\}$	$\{d, \varepsilon\}$	$\{e, \varepsilon\}$

Calcul des *suivants*

$Suiv(S) := \{\dashv\}$; **pour** $A \in V - \{S\}$ $Prem(A) := \emptyset$
répéter
  pour chaque $A \rightarrow X_1 \cdots X_n$ $(n \geq 0)$
    suiv := $Suiv(A)$
    pour $i = n \cdots 1$
      ajouter suiv à $Suiv(X_i)$
      si $\varepsilon \in Prem(X_i)$ **alors** ajouter $Prem(X_i) - \{\varepsilon\}$ à suiv
      sinon suiv := $Prem(X_i)$
tant que un $Suiv(X_i)$ a été modifié

Exemple de calcul des suivants (pour les non-terminaux)

avec $\mathcal{G}_2$: $S \rightarrow BABAC, A \rightarrow a, B \rightarrow b \mid DC, C \rightarrow c \mid E, D \rightarrow d \mid \varepsilon, E \rightarrow e \mid \varepsilon$

- initialement $Suiv(S) = \{\dashv\}$
- $S \rightarrow BABAC$
  - $Suiv(C) = Suiv(S) = \{\dashv\}$,
  - $Suiv(A) = Prem(C) \cup Suiv(C) = \{c, e\} \cup \{\dashv\}$,
  - $Suiv(B) = Prem(A) = \{a\}$,
  - $Suiv(A) \cup= Prem(B) \cup Suiv(B) = \{c, e, \dashv\} \cup \{b, d, c, e\} \cup \{a\}$
- $B \rightarrow DC$
  - $Suiv(C) \cup= Suiv(B) = \{\dashv\} \cup \{a\}$
  - $Suiv(D) = Prem(C) \cup Suiv(C) = \{c, e\} \cup \{\dashv, a\}$
- $C \rightarrow E$
  - $Suiv(E) = Suiv(C) = \{\dashv, a\}$

Conflits LL(1)

- si la grammaire n'est pas LL(1), il y aura des *conflits*
  - soit parce que dans un choix $A \rightarrow \alpha \mid \beta$, $Prem(\alpha) \cap Prem(\beta) \neq \emptyset$ (comme on l'a vu c'est notamment le cas si la grammaire contient des récursivités gauches)
  - soit α est annulable, et $Suiv(A) \cap Prem(\beta) \neq \emptyset$
- comment résoudre les conflits LL(1)
  - commencer par regarder s'il y a des récursivités gauches, et transformer la grammaire pour les enlever
  - factoriser les préfixes communs des règles : $A \rightarrow \alpha\beta \mid \alpha\gamma$ devient $A \rightarrow \alpha A', A' \rightarrow \beta \mid \gamma$
  - voir si l'on peut remonter des préfixes communs produits par les parties droites : dans $A \rightarrow B\beta \mid C\gamma, B \rightarrow \alpha\sigma, C \rightarrow \alpha\rho$, remonter α dans les règles de A et factoriser
  - les productions du vide peuvent aussi donner de nombreux conflits

Problèmes de conflits LL(1)

- par exemple pour un sous-ensemble de la grammaire de Java pour la désignation d'attributs/méthodes : `a.m(...).b.c`) :

```
fact ::= appel | variable | ...
variable ::= prefixe var
appel ::= prefixe app
prefixe ::= [ THIS "." ] ( ( var | app ) "." )*
var ::= IDENT
app ::= IDENT args
args ::= "(" [ arguments ] ")"
```

- il est clair que *variable* et *appel* commençant tous 2 par un préfixe, la grammaire n'est pas LL(k) (la longueur d'un préfixe n'est pas bornée)
- idem pour *var* et *app* (mais ici la grammaire est LL(2) car il suffit d'aller regarder s'il y a une parenthèse après l'identificateur)

Résolution de conflits LL(1)

- on peut essayer de rendre la grammaire LL(1) en factorisant : syntaxiquement, *app(el)* est une *var(iable)* suivie d'arguments

```
fact ::= prefixe var [ args ]
prefixe ::= [ THIS "." ] ( var [ args ] "." )*
```

- on a avancé, mais ce n'est pas fini : comment choisir entre le *var* du préfixe et celui de *fact* puisque *prefixe* est annulable ?
  - regarder s'il y a un `"."` qui suit ne suffit pas car les arguments peuvent contenir `"."` (par ex. `m(1,a.b,2)` n'a pas de préfixe)
- donc on tansforme les règles :

```
fact ::= [ THIS "." ] var [ args ] ( "." var [ args ] )*
```

- et si un facteur peut être seulement `this`, alors

```
fact ::= THIS ( "." var [ args ] )*
       |  var [ args ] ( "." var [ args ] )*
```

$Prem(THIS) \cap Prem(var) = \emptyset$, et le `.` décide s'il faut continuer dans l'* : cette partie de la grammaire est LL(1)

Principes de la méthode par descente récursive

- chaque règle EBNF donne lieu à une fonction

  par exemple pour la grammaire des expressions arithmétiques
  ({...} signifie 0 ou plusieurs fois)

$$E \rightarrow T\ \{\ (+|\text{-})\ T\ \} \qquad T \rightarrow F\ \{\ (*|/)\ F\ \} \qquad F \rightarrow \textbf{ident} \mid (\ E\ )$$

(`sy` est la fenêtre, `next()` avance sur le prochain symbole d'entrée)

```
void E() { T();
           while (sy == '+' || sy == '-') { next(); T(); }
         }
void T() { F();
           while (sy == '*' || sy == '/') { next(); F(); }
         }
void F() { if (sy == ident) next();
           else if (sy == '(') { next(); E();
                                 if (sy == ')') next(); else error();
                               }
           else error();
         }
```

JavaCC : un générateur d'analyseurs lexico-syntaxique descendants en *Java*

- permet d'écrire un analyseur syntaxique récursif descendant ; la grammaire – fortement LL(1) par défaut – est décrite en EBNF, et inclut la définition de l'analyseur lexical
- permet d'indiquer comment traiter les aspects non LL(1) de la grammaire
- permet d'indiquer des actions sémantiques en *Java*
- gère des attributs hérités et synthétisés (grammaire attribuée à gauche – *LAG*) pour la sémantique
- fournit des outils utiles (documentation de la grammaire, construction de l'arbre syntaxique, couverture de la grammaire, options de mise au point)

Forme d'une définition de grammaire pour *JavaCC*

- grammaire-*JavaCC*=

  options
  `PARSER_BEGIN` (*identificateur*)
  unité-de-compilation (une classe en Java)
  `PARSER_END` (*identificateur*)
  (production)*

- mots réservés

`EOF`	`IGNORE_CASE`	`JAVACODE`	`LOOKAHEAD`
`MORE`	`PARSER_BEGIN`	`PARSER_END`	`SKIP`
`SPECIAL_TOKEN`	`TOKEN`	`TOKEN_MGR_DECLS`	

Options par défaut de *JavaCC*

```
options {
  LOOKAHEAD = 1;
  CHOICE_AMBIGUITY_CHECK = 2;
  OTHER_AMBIGUITY_CHECK = 1;
  STATIC = true;
  DEBUG_PARSER = false;
  DEBUG_LOOKAHEAD = false;
  DEBUG_TOKEN_MANAGER = false;
  ERROR_REPORTING = true;
  JAVA_UNICODE_ESCAPE = false;
  UNICODE_INPUT = false;
  IGNORE_CASE = false;
  USER_TOKEN_MANAGER = false;
  USER_CHAR_STREAM = false;
  BUILD_PARSER = true;
  BUILD_TOKEN_MANAGER = true;
  SANITY_CHECK = true;
  FORCE_LA_CHECK = false;
}
```

Définition de la classe de l'analyseur de *JavaCC*

```
PARSER_BEGIN(MonAnalyseur)
  public class MonAnalyseur {
    // l'analyseur produit sera dans MonAnalyseur.java
    public static void main (String args []) {
      MonAnalyseur parser = new MonAnalyseur (System.in);
      try {
        parser.axiome ();
      } catch (ParseException e) {
        System.out.println ("erreur : " + e.getMessage ());
      }
    }
  }
PARSER_END(MonAnalyseur)
// les productions
...
void axiome () : { } { ... }
                        // pas nécessairement la 1ère règle
...
```

Les productions en *JavaCC*

Il y a 4 types de productions

- les productions : la grammaire du langage proprement dite
- les productions d'expressions régulières : pour définir les unités lexicales du langage, comme vu précédemment
- les déclarations de l'analyseur lexical : pour définir certaines propriétés de l'analyseur lexical
- les productions en code *Java* : permettent de "programmer" certaines règles de la grammaire (à ses risques et périls)

On voit en particulier que l'analyseur lexical et l'analyseur syntaxique sont définis ensemble

Les productions de la grammaire pour *JavaCC*

- chaque production (règle) de la grammaire produira une méthode de la classe *MonAnalyseur*
  - méthode qui peut avoir un résultat, des paramètres et des variables locales
- production = *prototype-de-méthode* : *bloc* { règles }
- règles = règle (| règle) *
- règle = (terme-de-règle) *
- terme-de-règle =
  - **(** règles **)** [+ | * | **?**]
  - | **[** règles **]**
  - | [*variable* =] expression-régulière
  - | [*variable* =] *appel-de-méthode*
  - | *bloc*
  - | résolution-de-conflit

Exemple de productions *JavaCC*

La déclaration de classes *Java*

```
void ClassDeclaration() :
{}
{
  ( "abstract" | "final" | "public" )*
  UnmodifiedClassDeclaration()
}

void UnmodifiedClassDeclaration() :
{}
{
  "class" <IDENT> [ "extends" Name() ]
  [ "implements" NameList() ]
  ClassBody()
}
```

`<IDENT>` est le nom d'une expression régulière définie auparavant comme *token*

Exemple de productions *JavaCC* avec paramètres (1)

Des déclarations simples en C : on compte la taille des variables

(l'entrée `int a[2][3]` affichera `a : array of 2 array of 3 int --> 24`)

```
void decls () :
{ String s; int t; /* nom et taille du type */ }
{
  ( <INT> {t = 4;} | <FLOAT> {t = 8;} ) {s = token.image;}
  variable (s, t) ( "," variable (s, t) )* ";"
}

void variable (String s, int t) :
{ int nb = 1; }
{
  <IDENT> {System.out.print (token.image + " : ");}
  [ nb = dimensions (1) ]
  { System.out.println (s + " --> taille = " + t * nb); }
}
```

Exemple de productions *JavaCC* avec paramètres (2)

```
int dimensions (int nb) :
{ int v; int n; }
{
   "["
   <CONST>
     { try {
         v = Integer.parseInt (token.image);
       } catch (NumberFormatException e) {
         v = 0;
       }
       System.out.print ("array of " + v + " ");
       n = nb * v;
     }
   "]"
   [ n = dimensions (n) ]
   { return n; }
}
```

Les productions d'expressions régulières en *JavaCC*

production-expression-régulière =
[états-lexicaux]
genre [[IGNORE_CASE]] :
{ définition (| définition) * }

états-lexicaux = /* *analogue aux start conditions de flex* */
<*>
| < *identificateur* (, *identificateur*)* >

genre = TOKEN | SPECIAL_TOKEN | SKIP | MORE

définition = expression-régulière [*bloc*] [: *identificateur*]

expression-régulière =
chaîne
| < [[#] *identificateur* :] expression-régulière-classique
| < *identificateur* >
| < EOF >

Exemples de productions d'expressions régulières

- non mémorisé, non retourné à l'analyseur syntaxique

```
SKIP : { " " | "\t" | "\n" }
```

- mémorisé, et retourné à l'analyseur syntaxique

```
TOKEN : {
    < ABSTRACT: "abstract" >
  | < BOOLEAN: "boolean" >
  | < BREAK: "break" >
  ...
}
TOKEN : {
    < INTEGER_LITERAL :
        <DECIMAL> (["l","L"])?
      | <OCTAL> (["l","L"])?
    >
  | < #DECIMAL: ["1"-"9"] (["0"-"9"])* >
  | < #OCTAL: "0" (["0"-"7"])* >
  ...
}
```

MORE, SPECIAL_TOKEN et utilisation d'états

Pour reconnaître les commentaires de la forme `/* ... */`

- dans l'état par défaut, sur un début de commentaire, on passe dans un état ad-hoc

```
MORE : { "/*" : IN_COMMENT }
```

- dans l'état `IN_COMMENT`, si on rencontre `*/`, c'est la fin du commentaire ; on repasse en mode par défaut, en ayant cumulé tout le texte du commentaire (à cause des `MORE`).

```
<IN_COMMENT> SPECIAL_TOKEN :
  { <COMMENT: "*/" > : DEFAULT }
```

- on reconnaîtra n'importe quel caractère jusqu'à la fin du commentaire (sauf `*/`)

```
<IN_COMMENT> MORE : { < ~[] > }
```

- le commentaire n'est pas retourné comme symbole terminal à l'analyseur syntaxique, mais il sera chaîné au prochain terminal qui sera reconnu (`SPECIAL_TOKEN`)

Les productions en code *Java*

- utilisées lorsque la définition en EBNF d'une partie de la grammaire n'est pas aisée, ou pour effectuer une reprise sur erreur

  production-en-code-*Java* = `JAVACODE` *prototype et bloc de méthode*

  exemple :

```
JAVACODE void AllerAuPointVirgule () {
// code pour avancer jusqu'au prochain ';'
}
```

- *JavaCC* ne peut pas analyser le code et considère ces productions comme une *boîte noire* ; en particulier, il ne peut pas connaître l'ensemble *Prem* de cette production

```
void UnNonTerminal () : { }
{   AllerAuPointVirgule ()
  | UnAutreNonTerminal () ... ";"
}
```

  La règle `AllerAuPointVirgule` sera systématiquement invoquée `:-(`

Points de conflits potentiels et résolution par défaut

- règles $A = (\beta \mid \gamma)$: si $Prem(\beta) \cap Prem(\gamma) \neq \emptyset$
  β est choisi pour tout symbole de fenêtre dans $Prem(\beta)$

- règles $A = \beta (\gamma)^+ \delta$ ou $A = \beta (\gamma)^* \delta$: si
  $Prem(\gamma) \cap Prem(\delta) \neq \emptyset$
  γ est choisi tant que le symbole de fenêtre est dans $Prem(\gamma)$

- règles $A = \beta (\gamma)? \delta$ ou $A = \beta [\gamma] \delta$: si
  $Prem(\gamma) \cap Prem(\delta) \neq \emptyset$
  γ est choisi si le symbole de fenêtre est dans $Prem(\gamma)$

Résolution des conflits

en augmentant la taille de la fenêtre

- soit la grammaire

```
void T() : { } { ( X() | Y() | "z" Z() ) }
void X() : { } { "t" "x" }
void Y() : { } { "t" "y" }
void Z() : { } { ( X() )+ Y() }
```

- c'est le symbole après `t` qui permet de choisir entre `X` et `Y` dans `T`, et permet de boucler ou non dans `Z` : la grammaire est LL (2)
- il est possible de donner l'option globale `lookahead=2` : solution coûteuse en général
- on peut indiquer que la grammaire est localement LL(2)

```
void T() : { } { ( LOOKAHEAD(2) X() | Y() | "z" Z() ) }
void Z() : { } { ( LOOKAHEAD(2) X() )+ Y() }
```

- ... mais on peut aussi transformer la grammaire

Grammaires non LL(k)

résolution du conflit en donnant une expression régulière

- soit la grammaire non LL(k), pour k aussi grand que possible

```
void T() : { } { ( X() | Y() | "z" Z() ) }
void X() : { } { ("t")+ "x" }
void Y() : { } { ("t")* "y" }
void Z() : { } { ( X() )+ Y() }
```

- on pourrait indiquer un k très grand, en croisant les doigts pour ne jamais rencontrer de suite de `t` aussi longue
- ou on ne prévoit pas de limite à k

```
void T():{} {(LOOKAHEAD(("t")+ "x") X() | Y() | "z" Z())}
void Z():{} {(LOOKAHEAD(("t")+ "x") X() )+ Y()}
```

- l'analyseur lexical "avalera" la suite de `t` : si elle est suivie d'un `x`, il fera le choix de `X` et sinon d'un `Y`, puis l'analyse syntaxique reprendra sur le premier `t` de la suite
  (on a donc un retour en arrière d'un nombre non borné de `t`, ce qui ne garantit plus la linéarité de l'analyseur)

Encore plus fort ! Résolution syntaxique

- les conflits ne peuvent pas toujours être résolus à l'aide d'expressions régulières
- par exemple un langage où `id (exp)` peut être une variable indicée ou un appel de fonction

```
void Stat(): {} { LOOKAHEAD ((Des() ":=")) Ass() | Call()
void Ass() : {} { Des() ":=" Exp() ";" }
void Call(): {} { "i" "(" Args() ")" ";"  }
void Des() : {} { "i" ( "(" Exp() ")" )* }
void Exp() : {} { Des() ( "+" Des() )* {print("-->Exp");}
void Args(): {} { Exp() ( "," Exp() )* }
```

- les action sémantiques ne sont pas effectuées lors de la résolution de conflit

Toujours plus fort ! Résolution sémantique

- les conflits ne peuvent pas toujours être résolus par la syntaxe : si on étend le langage précédent

```
void Des(): {} { ( Var() | Call() ) }
void Var(): {} { <ID> ( "(" Exp() ")" )* }
```

`v := x(i)` est syntaxiquement ambigu : `x(i)` est une variable si `x` est une variable et un appel si `x` est une fonction, d'où besoin d'un *prédicat sémantique* (méthode `IsVar` ci-dessous)

```
void Stat(): {} { LOOKAHEAD({IsVar()}) Ass() | Call() }
void Des() : {} { LOOKAHEAD({IsVar()}) Var() | Call() }
```

- ce qui suppose l'existence d'un dictionnaire (et donc du traitement sémantique des déclarations)

```
private static boolean IsVar () {
  Token t = getToken (1); Object o = dict.get (t.image);
  return (o != null && (String) o == "var");
}
```

Forme générale de résolution des conflits

- les trois genres de résolution peuvent être présents simultanément

résolution-des-conflits =
`LOOKAHEAD(` [*constante-entière*] [`,`]
[règles] [`,`]
[{*expression-booléenne*}]
)

- au moins une des trois composantes doit être présente, et s'il y en a plus d'une, elles doivent être séparées par des virgules
- la limite de symboles examinés n'est pas prise en compte pour la résolution sémantique
- la résolution sémantique semble emporter la décision sur les autres

L'analyseur lexical produit par *JavaCC*

- l'analyseur lexical est une classe appelée `MonAnalyseurTokenManager`
  - `Token getNextToken () throws TokenMgrError` : la méthode qui reconnaît un des symboles définis comme `TOKEN` dans la grammaire
- un symbole est un objet de la classe `Token` qui possède notamment :
  - `int kind` : le code interne du symbole, manipulable au travers des noms symboliques dans les expressions régulières

    ```
    < PLUS : "+" >
    ```

    ces noms symboliques sont définis dans la classe `MonAnalyseurConstants`
  - `String image` : la chaîne qui a été reconnue

L'analyseur syntaxique produit par *JavaCC*

- il est produit dans la classe `MonAnalyseur` (`MonAnalyseur.java`)
  - chaque production de la grammaire donne une méthode
  - le constructeur `MonAnalyseur(java.io.InputStream str)` crée l'analyseur lexical qui lira sur `str` (disponible si les options `USER_TOKEN_MANAGER` et `USER_CHAR_STREAM` sont **toutes deux fausses** – ce qui est le cas par défaut)
- il est possible d'appeler des méthodes de l'analyseur lexical, notamment
  - `Token getNextToken() throws ParseException` : "avale" le prochain symbol, à ne pas confondre avec :
  - `Token getToken(int x) throws ParseException`, qui retourne le x-ième symbole après le symbole courant
- et d'utiliser l'attribut `Token token` : le symbole courant, équivalent à `getToken(0)`

Les options pour résoudre les conflits

- `LOOKAHEAD = k`
  - nombre maximum de symboles terminaux à examiner en général pour prendre une décision (1 par défaut – grammaire LL (1))
  - augmenter ce nombre diminue l'efficacité de l'analyseur et augmente sa taille
  - ce nombre peut être modifié localement, ce qui est un bon compromis
- `force_la_check` : permet d'effectuer les vérifications sur les conflits dans la grammaire (ce qui n'est pas fait par défaut quand l'utilisateur donne des indications pour résoudre les conflits – `LOOKAHEAD=2` par exemple – et se trompe)

D'autres options utiles

- `static` : par défaut, l'analyseur produit est une classe avec des méthodes statiques ; en ce cas, il ne peut y avoir qu'un seul analyseur ; si l'option est fausse, les méthodes ne sont pas statiques et il est possible de créer plusieurs analyseurs
- `unicode_input` : l'analyseur lexical travaille par défaut sur des textes ASCII mais peut travailler sur des textes Unicode si cette option est vraie
- `java_unicode_escape` : permet d'obtenir un analyseur lexical reconnaissant les séquences unicode (défaut = false)
- `ignore_case` : l'analyseur lexical peut différencier les lettres minuscules des majuscules (option par défaut) ou non (si l'option est vraie)

Traitement des erreurs

- une erreur syntaxique lève une exception `ParseException`, tandis qu'une erreur lexicale lève une `TokenMgrError`
- il est possible d'indiquer une liste d'exceptions (en plus des deux précédentes) dans une règle :

```
void R () throws X1, X2 : { ... } { ... }
```

- récupération en mode panique

```
void R () {} { R1() | R2() | AvancerJusque(T) }
```

où `AvancerJusque` est une production `JAVACODE` qui avance (par `getNextToken`) jusqu'à lire le symbole `T`, et qui sera appelée si le symbole courant n'est pas dans les premiers de `R1` ou `R2`

- non satisfaisant si l'erreur a lieu dans `R1` ou `R2`

Récupération des erreurs “profondes”

- une meilleure solution est :

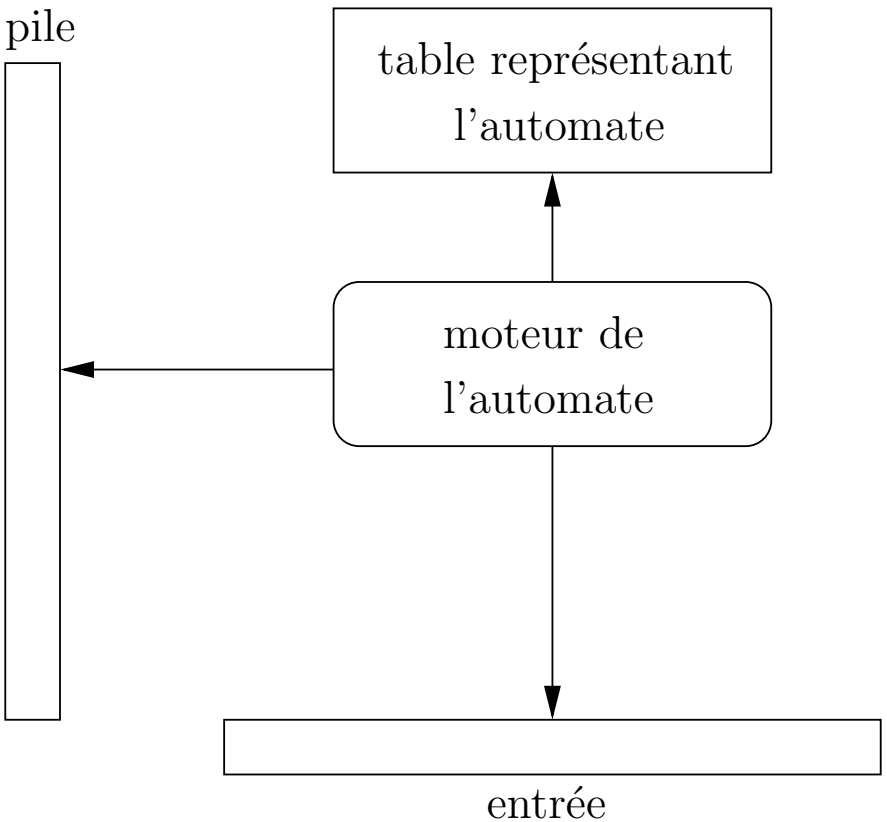
```
void R () {} { try {
                    ( R1() | R2() ) <T>
                  } catch (ParseException e) {
                      AvancerJusque(T)
                  }
                }
```

avec

```
private static void AvancerJusque (int sy) {
  while (token.kind != sy && token.kind != EOF)
    token = getNextToken ();
}
```

Analyse descendante par automate à pile

- l'analyse descendante peut aussi être effectuée à l'aide d'un automate, dont la pile se substitue à la pile des appels de fonction de la méthode par descente récursive



Principes de l'analyse descendante par automate à pile

- si $S \underset{\text{lm}}{\Rightarrow}^* w\rho$, et que w a été reconnu, la pile contient $\bar{\rho}$: le premier symbole de ρ est en sommet de pile, et c'est lui qu'on doit d'abord reconnaître

- la table M est une matrice $N' \times T'$: pour un non-terminal A et une fenêtre f, $M[A, f]$ donne la règle $A \rightarrow \alpha$ qu'il faut reconnaître (si $M[A, f]$ est vide, f n'est pas une fenêtre valide pour $A \Rightarrow$ erreur)

Construction de la table de l'automate fortement LL(1)

pour tout $A \rightarrow \alpha$
  pour tout $a \in Prem(\alpha)$ ajouter $A \rightarrow \alpha$ à $M[A, a]$
  si α est annulable
    pour tout $a \in Suiv(A)$ ajouter $A \rightarrow \alpha$ à $M[A, a]$

la grammaire n'est pas fortement LL(1) s'il existe une entrée $M[A, a]$ avec plus d'une action

la table pour $\mathcal{G}_2$ $S \rightarrow BABAC, A \rightarrow a, B \rightarrow b \mid DC, C \rightarrow c \mid E, D \rightarrow d \mid \varepsilon, E \rightarrow e \mid \varepsilon$

	a	b	c	d	e	$\dashv$
S	$S \rightarrow BABAC$	$S \rightarrow BABAC$	$S \rightarrow BABAC$	$S \rightarrow BABAC$	$S \rightarrow BABAC$	
A	$A \rightarrow a$					
B	$B \rightarrow DC$	$B \rightarrow b$	$B \rightarrow DC$	$B \rightarrow DC$	$B \rightarrow DC$	
C	$C \rightarrow E$		$C \rightarrow c$		$C \rightarrow E$	$C \rightarrow E$
D	$D \rightarrow \varepsilon$		$D \rightarrow \varepsilon$	$D \rightarrow d$	$D \rightarrow \varepsilon$	$D \rightarrow \varepsilon$
E	$E \rightarrow \varepsilon$				$E \rightarrow e$	$E \rightarrow \varepsilon$

Fonctionnement de l'automate

- le moteur de l'automate fonctionne de la manière suivante : (avec X le sommet de pile et f la fenêtre)
  - initialement, la pile contient $\dashv S$
  - si X est un terminal t et que $f = t$, on dépile et on avance (sinon la phrase est incorrecte : $f \notin \mathit{Prem}(\rho)$)
    si $t = f = \dashv$, l'analyse termine : la phrase a été reconnue
  - si X est un non-terminal A et que $M[A, f] = A \rightarrow \alpha$, dépiler A et empiler $\bar{\alpha}$ (si $M[A, f]$ est vide, c'est une erreur : $S' \underset{\mathrm{lm}}{\Rightarrow}^* w\rho \underset{\mathrm{lm}}{\Rightarrow} wA\rho'$, et $f \notin \mathit{Prem}(A\rho')$)

Exemple de fonctionnement de l'automate à pile

- avec l'entrée *badac* et la grammaire $\mathcal{G}_2$
  $S \to BABAC, A \to a, B \to b \mid DC, C \to c \mid E, D \to d \mid \varepsilon, E \to e \mid \varepsilon$

dérivation	pile	fenêtre	action
$S' \underset{\text{lm}}{\Rightarrow} S\dashv$	$\dashv S$	b	$S \to BABAC$
$\underset{\text{lm}}{\Rightarrow} BABAC\dashv$	$\dashv CABAB$	b	$B \to b$
$\underset{\text{lm}}{\Rightarrow} bABAC\dashv$	$\dashv CABAb$	b	dépiler et avancer
$\cdots ABAC\dashv$	$\dashv CABA$	a	$A \to a$
$\underset{\text{lm}}{\Rightarrow} \cdots aBAC\dashv$	$\dashv CABa$	a	dépiler et avancer
$\cdots BAC\dashv$	$\dashv CAB$	d	$B \to DC$
$\underset{\text{lm}}{\Rightarrow} \cdots DCAC\dashv$	$\dashv CACD$	d	$D \to d$
$\underset{\text{lm}}{\Rightarrow} \cdots dCAC\dashv$	$\dashv CACd$	d	dépiler et avancer
$\cdots CAC\dashv$	$\dashv CAC$	a	$C \to \varepsilon$
$\underset{\text{lm}}{\Rightarrow} \cdots AC\dashv$	$\dashv CA$	a	$A \to a$
$\underset{\text{lm}}{\Rightarrow} \cdots aC\dashv$	$\dashv Ca$	a	dépiler et avancer
$\underset{\text{lm}}{\Rightarrow} \cdots C\dashv$	$\dashv C$	c	$S \to c$
$\underset{\text{lm}}{\Rightarrow} \cdots c\dashv$	$\dashv c$	c	dépiler et avancer
$\underset{\text{lm}}{\Rightarrow} \cdots \dashv$	$\dashv$	$\dashv$	succès

Automate LL(k) et automate fortement LL(k)

- nous pouvons voir que dans la construction fortement LL(1), le contexte dans lequel un non-terminal est attendu n'intervient pas
- donc fortement-LL(k) $\subset$ LL(k)
  par exemple pour $\mathcal{G}_1$: $S \rightarrow aAa \mid bAba, A \rightarrow b \mid \varepsilon$ qui comme nous l'avons vu est LL(2) mais pas fortement LL(2)
  - $Prem(A) = \{b, \varepsilon\}$ et $Suiv_2(A) = \{a\dashv, ba\}$: la fenêtre ba prédit $A \rightarrow b$ pour $S \rightarrow aAa$ ($ba \in 2 : Prem(A)Suiv_2(A)$) et $\mathcal{A} \rightarrow \varepsilon$ pour $S \rightarrow bAba$ ($ba \in Suiv_2(A)$)
- pour construire l'automate LL(k), il faut donc restreindre les suivants de A au contexte dans lequel on peut trouver A

Principes de fonctionnement de l'automate LL(k)

- la pile ne contient plus des symboles de la grammaire, mais des *états*
- les règles de réécriture seront notées $\pi \mathsf{I} \omega \models \pi' \mathsf{I} \omega'$
  comprendre : si la pile contient (en haut) la séquence d'états π et que la fenêtre commence par w, alors remplacer π par π' et w par w'
- exemple
  - avancer dans le texte sur t et empiler l'état q s'écrira $\mathsf{I}t \models q\mathsf{I}$
- le générateur d'analyseur produira une table comprenant les règles de réécriture nécessaires à l'analyse
- un item de la forme $[A \rightarrow \alpha \cdot \beta, w]$ représentera un état q : on est en train de reconnaître un A, qui dans ce contexte est suivi de w ($|w| = k$) ; on a déjà reconnu α et on s'apprête à reconnaître β

Construction de l'automate LL(k)

soit Q l'ensemble des états, R l'ensemble des règles

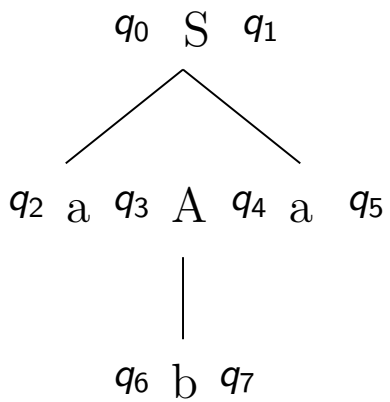
soit $q_0 = [S' \rightarrow \cdot S, \dashv^k]$; $Q := \{q_0\}$; $R := \emptyset$

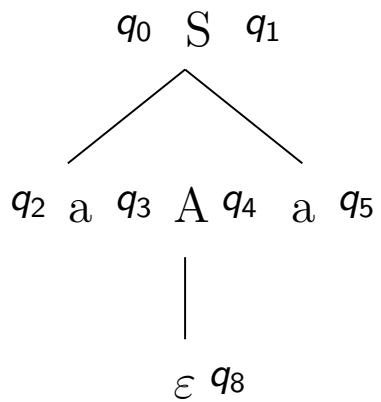
pour $q \in Q$, q pas encore traité
  si $q = [A \rightarrow \alpha \cdot, w]$ **alors**
    $R := R \cup \{q\mathsf{I} \models \mathsf{I}\}$ (donc simplement dépiler q)
  sinon si $q = [A \rightarrow \alpha \cdot t\beta, w]$ **alors**
    avec $q' = [A \rightarrow \alpha t \cdot \beta, w]$: $Q := Q \cup \{q'\}$; $R := R \cup \{q\mathsf{I}t \models q'\mathsf{I}\}$
      (on avance et on note qu'on a avancé)
  sinon $(q = [A \rightarrow \alpha \cdot B\beta, w])$
    avec $q' = [A \rightarrow \alpha B \cdot \beta, w]$: $Q := Q \cup \{q'\}$
    pour tout $q'' = [B \rightarrow \cdot \gamma, w']$ **tel que** $B \rightarrow \gamma \in P$, $w' = Prem_k(\beta w)$
      $Q := Q \cup \{q''\}$
      avec $w'' = Prem_k(\gamma\beta w)$: $R := R \cup \{q\mathsf{I}w'' \models q'q''\mathsf{I}w''\}$

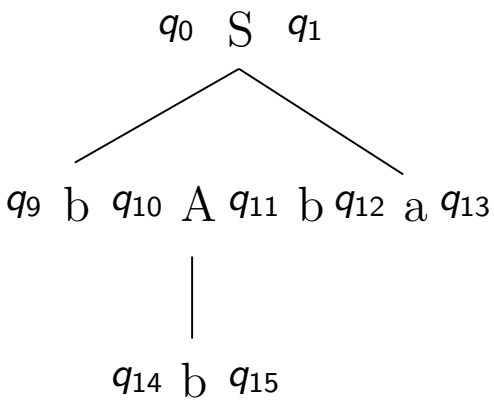
l'automate s'arrête dans la configuration $q_1\mathsf{I}\dashv$, $q_1 = [S' \rightarrow S \cdot, \dashv^k]$

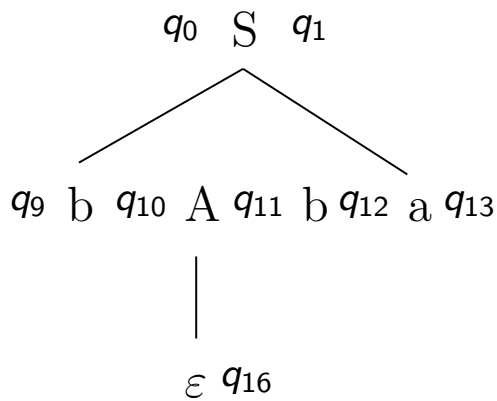
Item LL(k) et positions dans les arbres

pour $\mathcal{G}_1$: $S \rightarrow aAa \mid bAba, A \rightarrow b \mid \varepsilon$









Actions de l'automate LL(2)

pour $\mathcal{G}_1$: $S \rightarrow aAa \mid bAba, A \rightarrow b \mid \varepsilon$

état	fenêtre	action
$q_0 = [S' \rightarrow \cdot S, \dashv\dashv]$	a	$q_0 \mid a \models q_1 q_2 \mid a,\ q_2 = [S \rightarrow \cdot aAa, \dashv\dashv]$
	b	$q_0 \mid b \models q_1 q_9 \mid b,\ q_9 = [S \rightarrow \cdot bAba, \dashv\dashv]$
$q_1 = [S' \rightarrow S \cdot, \dashv\dashv]$	$\dashv$	succès
$q_2 = [S \rightarrow \cdot aAa, \dashv\dashv]$	a	$q_2 \mid a \models q_3 \mid,\ q_3 = [S \rightarrow a \cdot Aa, \dashv\dashv]$
$q_3 = [S \rightarrow a \cdot Aa, \dashv\dashv]$	b	$q_3 \mid b \models q_4 q_6 \mid b,\ q_6 = [A \rightarrow \cdot b, a\dashv]$
	a	$q_3 \mid a \models q_4 q_8 \mid a,\ q_8 = [A \rightarrow \cdot, a\dashv]$
$q_4 = [S \rightarrow aA \cdot a, \dashv\dashv]$	a	$q_4 \mid a \models q_5 \mid,\ q_5 = [S \rightarrow aAa \cdot, \dashv\dashv]$
$q_5 = [S \rightarrow aAa \cdot, \dashv\dashv]$		$q_5 \mid \dashv \models \mid \dashv$
$q_6 = [A \rightarrow \cdot b, a\dashv]$	b	$q_6 \mid b \models q_7 \mid,\ q_7 = [A \rightarrow b\beta, a\dashv]$
$q_7 = [A \rightarrow b \cdot, a\dashv]$		$q_7 \mid \models \mid$
$q_8 = [A \rightarrow \cdot, a\dashv]$		$q_8 \mid \models \mid$

Actions de l'automate LL(2) – suite

pour $\mathcal{G}_1$: $S \rightarrow aAa \mid bAba, A \rightarrow b \mid \varepsilon$

état	fenêtre	action
$q_9 = [S \rightarrow \cdot bAba, \dashv\dashv]$	b	$q_9 \mathsf{I} b \models q_{10},\ q_{10} = [S\ Pb \cdot Aba, \dashv\dashv]$
$q_{10} = [S \rightarrow b \cdot Aba, \dashv\dashv]$	bb	$q_{10} \mathsf{I} bb \models q_{11} q_{14} \mathsf{I} bb,\ q_{14} = A \rightarrow \cdot b, ba]$
	ba	$q_{10} \mathsf{I} ba \models q_{11} q_{16} \mathsf{I} ba,\ q_{16} = A \rightarrow \cdot, ba]$
$q_{11} = [S \rightarrow bA \cdot ba, \dashv\dashv]$	b	$q_{11} \mathsf{I} b \models q_{12} \mathsf{I},\ q_{12} = [S \rightarrow bAb \cdot a, \dashv\dashv]$
$q_{12} = [S \rightarrow bAb \cdot a, \dashv\dashv]$	a	$q_{12} \mathsf{I} a \models q_{13} \mathsf{I},\ q_{13} = [S \rightarrow bAba \cdot, \dashv\dashv]$
$q_{13} = [S \rightarrow bAba \cdot, \dashv\dashv]$		$q_{13} \mathsf{I} \models \mathsf{I}$
$q_{14} = [A \rightarrow \cdot b, ba]$	b	$q_{14} \mathsf{I} b \models q_{15} \mathsf{I},\ q_{15} = A \rightarrow b \cdot, ba]$
$q_{15} = [A \rightarrow b \cdot, ba]$		$q_{15} \mathsf{I} \models \mathsf{I}$
$q_{16} = [A \rightarrow \cdot, ba]$		$q_{16} \mathsf{I} \models \mathsf{I}$

Exemple de fonctionnement de l'automate LL(2)

pour $\mathcal{G}_1$: $S \rightarrow aAa \mid bAba, A \rightarrow b \mid \varepsilon$

- pour l'entrée aa
  $q_0 \mid aa\dashv \models q_1 q_2 \mid aa\dashv \models q_1 q_3 \mid a\dashv \models q_1 q_4 q_8 \mid a\dashv \models q_4 \mid a\dashv \models q_5 \mid \dashv \models q_1 \mid \dashv$
- pour l'entrée aba
  $q_0 \mid aba\dashv \models q_1 q_2 \mid aba\dashv \models q_1 q_3 \mid ba\dashv \models q_1 q_4 q_6 \mid ba\dashv \models q_1 q_4 q_7 \mid a\dashv \models q_1 q_4 \mid a\dashv \models q_1 q_5 \mid \dashv \models q_1 \mid \dashv$
- pour l'entrée bba
  $q_0 \mid bba\dashv \models q_1 q_9 \mid bba\dashv \models q_1 q_{10} \mid ba\dashv \models q_1 q_{11} q_{16} \mid ba\dashv \models q_1 q_{11} \mid ba\dashv \models q_1 q_{12} \mid a\dashv \models q_1 q_{13} \mid \dashv \models q_1 \mid \dashv$
- pour l'entrée $bbba$
  $q_0 \mid bbba\dashv \models q_1 q_9 \mid bbba\dashv \models q_1 q_{10} \mid bba\dashv \models q_1 q_{11} q_{14} \mid bba\dashv \models q_1 q_{11} q_{15} \mid ba\dashv \models q_1 q_{11} \mid ba\dashv \models q_1 q_{12} \mid a\dashv \models q_1 q_{13} \mid \dashv \models q_1 \mid \dashv$