**Rappels :**

On classe les grammaires G = (VT , VN, R, S) en quatres grandes familles (ou types ou classes) numérotés de 0 à 3, de la plus large à la plus petite au sens de l’inclusion stricte.

Type 0 - aucune restriction. Les langages engendrés sont qualifiés de récursivement énumérables.

Type 1 - toute règle r de R est de la forme : **r = αXβ → αmβ** avec α, β ∈ V∗ ; X ∈ VN ; m ∈ V+.

Attention m ne peut être le mot vide ! Ces grammaires sont dites contextuelles ou dépendant du contexte (α et β représentant ce contexte). Le mot vide ne pouvant être

généré par ces grammaires, une exception existe : la règle S → ε peut exister à condition que S ne soit pas présente dans une partie droite d’une règle de production.

Exemple : le P garçon → le petit garçon ; la P N → la petite N ; N → fille.

Type 2 - toute règle r de R est de la forme : **r = X → α** avec α ∈ V∗ ; X ∈ VN.

Ces grammaires sont dites algébriques, ou indépendantes du contexte (“context-free”), ou grammaires de Chomsky, ou C-grammaires.

Exemple : P → (P)|ε|PP : une grammaire de parenthèses.

Type 3 - toute règle r de R est de la forme : **r = X → α** avec α ∈ VT VN ∪ VT ∪ {ε} ; X ∈ VN ;

Ces grammaires sont dites régulières, ou rationnelles, ou grammaires de Kleene, ou K-grammaires.

Exemple : P → 0|1E|2E| . . . |9E ; E → 0E| . . . |9E|ε : une grammaire régulière d’indices.

| Théorème : On note Li l’ensemble des langages engendrés par les grammaires de type  i. On a alors l’inclusion stricte : L3 ⊂ L2 ⊂ L1 ⊂ L0. |
| --- |

Langages réguliers - Propriétés et caractérisations :

| Théorème : Les 4 propositions suivantes sont équivalentes :   1. Le langage L est défini par une expression régulière ; 2. Le langage L est généré par une grammaire régulière ; 3. Le langage L est reconnu par un automate fini déterministe ; 4. Le langage L est reconnu par un automate fini non déterministe. |
| --- |

| Théorème : La famille des langages réguliers L3 est la plus petite famille de langages qui contient les langages finis et qui est fermée pour les opérations réunion, produit et étoile. |
| --- |

| Théorème : Le langage inverse, complémentaire d’un langage régulier est régulier. L’intersection de deux langages réguliers est régulier. |
| --- |

Langages algébriques - Propriétés et caractérisations :

L’ensemble des arbres de dérivation (ou arbres syntaxiques) associé à une grammaire G = (VT, VN, R, S), noté A(G) est un ensemble d’arbres étiquetés construits par le schéma d’induction suivant.

Univers : Ensemble de tous les arbres dont les nœuds sont étiquetés par des symbole de V ∪ {ε}.

Base : Ensemble de tous les arbres réduits à une unique racine étiquetée par un symbole de V ∪ {ε}.

Règles : Soit une règle de production quelconque X → y1y2 . . . yn avec X ∈ VN, yi ∈ V ∪ {ε}.

| Théorème : L’ensemble des dérivations gauches d’une grammaire algébrique G = (VT, VN, R, S) est équipotent à A(G) |
| --- |

| Théorème : Tout langage régulier est non ambigu. |
| --- |

| Théorème : La famille des langages algébriques L2 est fermée pour l’union, la concaténation, l’opération \*. |
| --- |

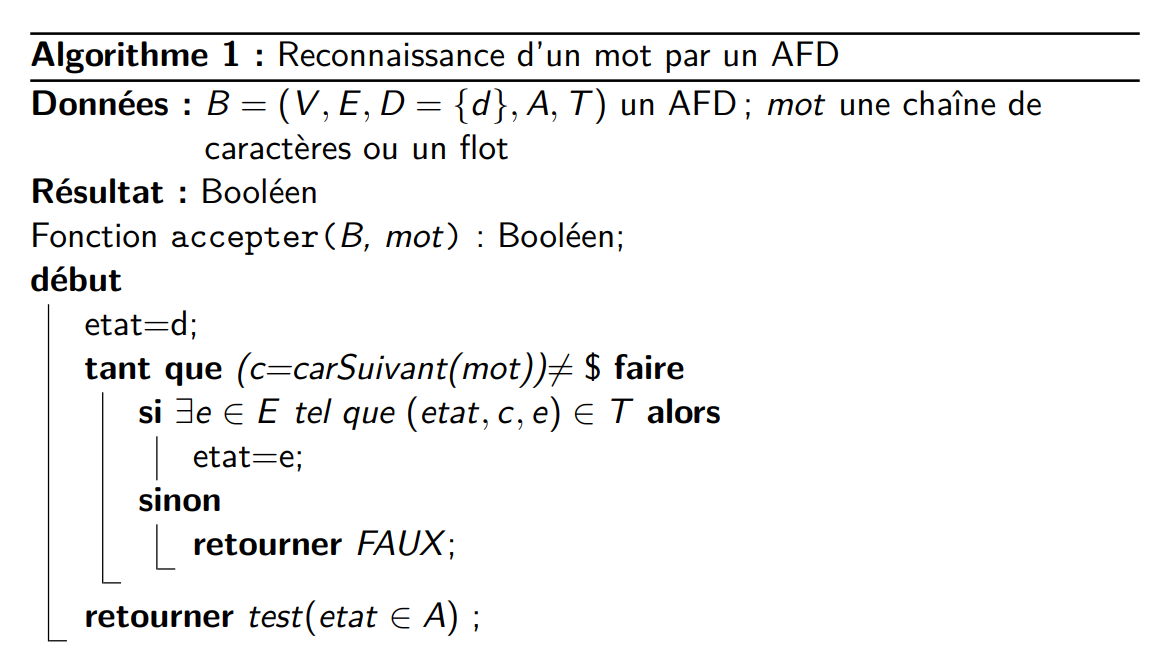
| Théorème : La famille des langages algébriques L2 n’est pas fermée pour l’intersection ni la complémentation. |
| --- |

Analyse du source :

1. lexicale : découpage en “jetons” (tokens) ;
2. syntaxique : vérification de la correction grammaticale et production d’une représentation intermédiaire (souvent un arbre) ;
3. sémantique : vérification de la correction sémantique du programme (contrôle de type (conversions), non déclarations, protection de composants (privé, public), . . . ).

L’analyse génère une table des symboles qui sera utilisée tout au long du processus de compilation. De plus, l’apparition d’erreurs dans chaque phase peut interrompre le processus ou générer des messages d'avertissement (warnings).

Remarque : L’analyse lexicale est souvent réalisée “à la demande” de l’analyse syntaxique, jeton par jeton. Ainsi la décomposition en phase (analyse lexicale, syntaxique, sémantique, . . . ) n’engendre pas forcément la même décomposition en “passes”, une passe correspondant à la lecture séquentielle du résultat de la phase précédente.



Analyse lexicale :

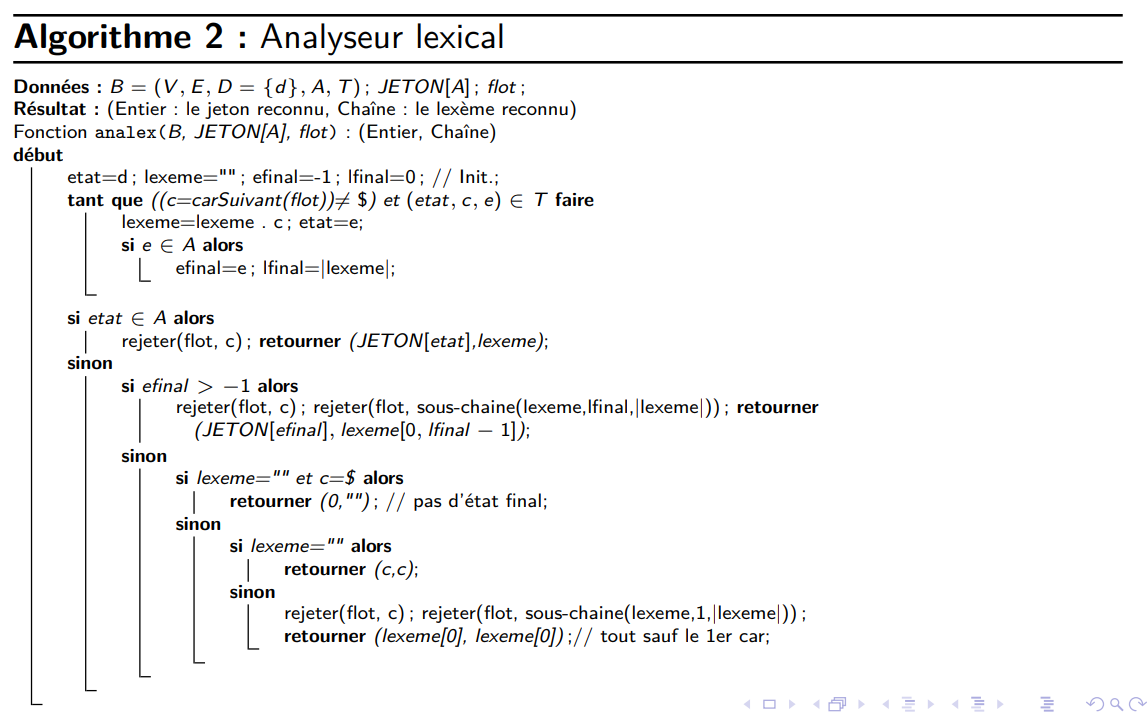
Rappels sur les AFD + Algo :

* un AFD possède un unique état initial
* l’ensemble des transitions peut être implémenté simplement par une table à double entrée TRANS[etatCourant][carCourant] qui contient l’état suivant
* aucun couple de transitions (ei, a, ej) , (ei, a, ek ) tels que j != k.

Implémentation des AFD : Voir diapo 28/29/30 (Fichier d'en-tête).

**Analyse lexicale :**

* Suite à la reconnaissance d’un mot ou lexème, l’analyseur lexical doit retourner un jeton (token) entier associé à la catégorie lexicale du mot accepté.
* Un jeton (token) est généralement représenté par un entier positif ou une instance de classe
* Les entiers inférieurs à 256 sont réservés aux mots clés composés d’un seul caractère : (“{”, “ ;”, “]”, ...). Leur code (ASCII, ISO Latin1, ...) correspondra ainsi à leur jeton
* Chaque mot clé de plus d’une lettre est également associé à son jeton : (if, 300), (else, 301), (while, 302), …
* On définira également un jeton pour chaque catégorie lexicale variable : (littéral entier, 303), (littéral chaîne, 304), …
* Pour les catégories lexicales variables, il faudra également “retourner” une valeur sémantique associée
* pour les littéraux entiers on pourrait retourner la valeur entière correspondante
* pour les identificateurs le lexème lui-même ou l’indice d’entrée correspondant dans la table des symboles
* De plus, un analyseur lexical doit reconnaître une suite de lexèmes dans un flot de caractères
* Dans l’automate d’états finis déterministe (AFD), chaque état terminal est associé à un jeton retournable
* C’est le chemin parcouru dans l’automate qui déterminera le jeton à retourner



Attention, si on a avancé dans l’AFD et que l’on se retrouve dans un état non terminal sans pouvoir avancer, il faudra reculer afin de retourner dans le dernier état terminal parcouru ! Ce recul nécessite de rejeter dans le flot d'entrées (ungetc) les caractères qui ont été lus en trop.

Pour l’algorithme suivant :

* La gestion des mots non reconnus est la suivante : retourner le jeton correspondant au code ASCII du premier caractère. Contrairement à cela, Lex lui ne retourne aucun jeton mais envoie ce premier caractère sur la sortie standard et tenter de se resynchroniser sur le caractère suivant ;
* on suppose dans cet algorithme que le symbole $ est retourné à l’infini par carSuivant() lorsqu’on est parvenu à la fin du flot ;
* Remarquons que dans le cas où l’état initial est également final, le mot vide est donc acceptable. Par conséquent, sur un mot non acceptable ou sur le mot vide, l’analyseur lexical retournera une suite infinie de jetons associés à l’état initial !

Implémentation de l’analyseur lexical : En C, une seule valeur pouvant être retournée par une fonction, on choisit de retourner le jeton et d’implémenter la valeur sémantique dans une variable globale lexème de type chaîne de caractères.

Une dernière fonctionnalité à réaliser par les analyseurs lexicaux est le filtrage des séparateurs (blancs : espaces, tabulations, ...) et des commentaires.

**Un langage et un outil pour l’analyse lexical - Flex :**

Lex est un outil permettant de générer un programme d’analyse lexicale à partir de définitions de modèles (expressions régulières) et d’actions à exécuter lors de la reconnaissance de ces modèles.

Exemple : L’analyseur lexical généré tente, de manière itérative, de reconnaître une expression régulière (pattern matching) puis exécute les instructions correspondantes. L’analyseur termine sur la fin de fichier (EOF) de l’entrée standard (CTRL-D pour le terminal). Les mots ne correspondant à aucune expression régulière sont rejetés dans la sortie standard sans aucun traitement particulier. Au cœur du source C lex.yy.c généré par flex, la fonction C : int yylex() d’analyse lexicale permet de retourner un jeton entier correspondant au modèle reconnu.

Dans l’exemple précédent, la fonction principale : int main() appelle yylex() itérativement jusqu’au caractère de fin de fichier.

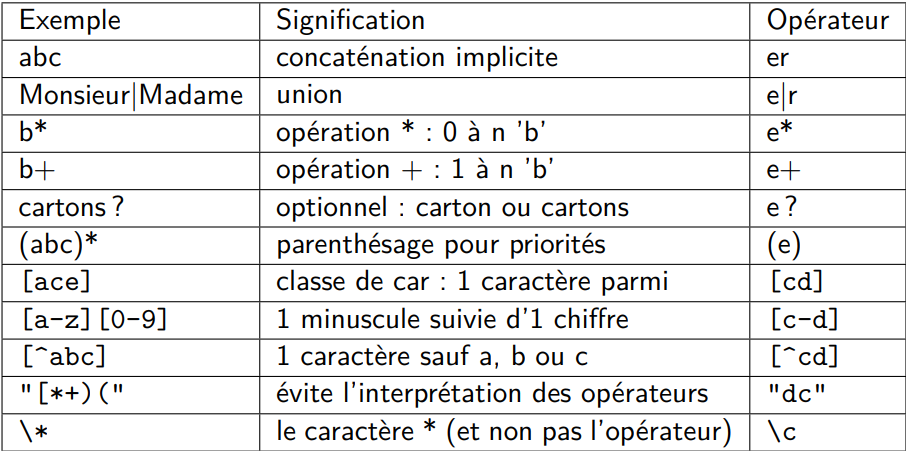
Si l’on observe le code C généré dans lex.yy.c, on s’aperçoit que l’automate fini déterministe calculé par flex est codé dans un tableau statique du programme C.

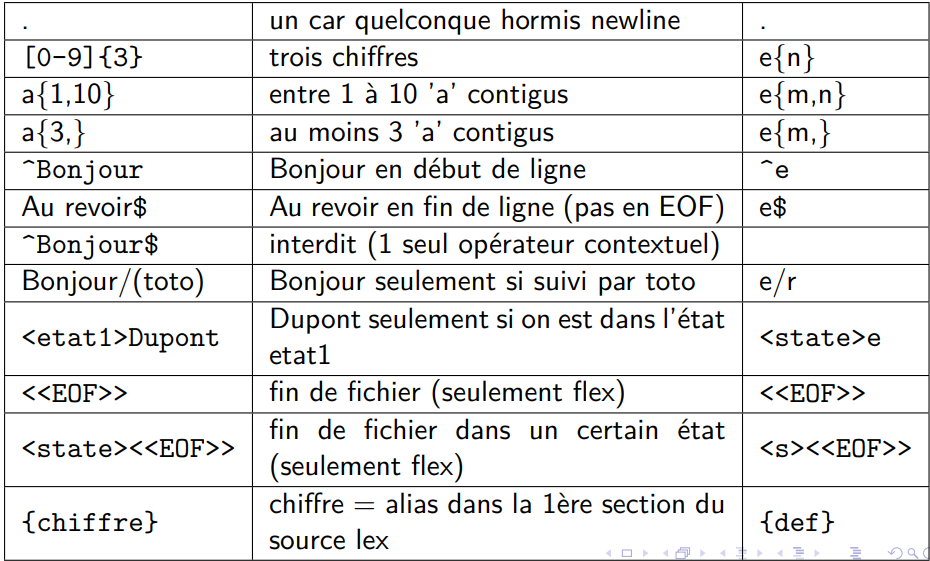
**Syntaxe et sémantiques des sources flex :**

Un source lex comprend 3 sections séquentielles :

* une section optionnelle de définitions. Elle contient les directives d’inclusions et les définitions globales C (variables, types, . . . ).
* Une section obligatoire de règles lex délimitée par %% au début. C’est la section centrale du source lex qui définit l’analyseur lexical en associant des instructions C à des expressions régulières.
* Une section optionnelle de fonctions C définies par l’utilisateur délimitée par %% au début. C’est là que l’on peut définir le main().

Syntaxe des expressions régulières :





La partie droite de chaque règle est un bloc d’instructions C. Le texte inclut entre accolades sera recopié intégralement dans lex.yy.c sans aucune analyse ni modification. En particulier, avec flex, on peut ne pas utiliser la librairie flex libfl.a à condition de définir la fonction principale main() ainsi que la fonction. Ne pas utiliser la librairie flex : %option noyywrap.

The %option noyywrap generally causes a lex compatible lexer generator (e.g. lex or flex) to emit a macro version of yywrap() that returns 1, which causes the lexer to stop lexing when the first end-of-file is reached.

Les instructions C peuvent référencer une variable :

* soit prédéfinie par lex : la chaîne **char\* yytext** de longueur int **yyleng** correspond au mot **reconnu dans le texte à analyser (lexème)**;
* soit définie en section définitions : dans ce cas, la variable est globale ;
* soit définie juste après l’accolade : dans ce cas, la variable est locale à la règle.

| **Variables prédéfinies :**  **yytext** **=** chaîne de car (char \*) contenant le lexème en cours de reconnaissance ;  **yyleng =** longueur (int) de yytext ;  **yyin =** flot d’entrée des caractères de type FILE\* (par défaut stdin) ; On peut rediriger le flot d’entrée sur le premier argument du main en faisant : yyin=fopen(argv[1],"r");  **yyout** **=** sortie standard de type FILE\*. Pour y afficher, faire : fprintf(yyout, "..."); |
| --- |

| **Fonctions prédéfinies :**   * **int yylex()** lit un lexème depuis le flot d’entrée et retourne le jeton associé. Retourne le jeton 0 pour finir. * **int input()** lecture d’un caractère depuis le flot d’entrée (yyinput en C++) ; input() équivaut à fgetc(yyin) ; * **void unput(int)** retour dans le flot d’entrée d’un car ; unput(c) équivaut à ungetc(c,yyin) * **int yywrap()** lorsque l’analyseur yylex() arrive en fin de fichier (EOF), il appelle yywrap(). Si yywrap retourne 1 (par défaut) alors yylex() retourne 0 (fin d’analyse). Si on voulait enchaîner sur un autre fichier, il faut redéfinir dans la section “définitions” du source lex, la fonction yywrap() afin qu’elle fasse pointer yyin sur le nouveau fichier puis retourne 0 ; * **yymore()** concatène dans yytext le prochain lexème avec celui en cours ; * **yyless(int n)** replace le lexème reconnu yytext dans le flot d’entrée à l’exception de ses n premiers caractères ; * **ECHO** affiche yytext ; ECHO équivaut à fprintf(yyout,yytext) ; * **REJECT** rejette le lexème reconnu dans le flot d’entrée et s’interdit de reconnaître la règle courante au prochain essai (appel de yylex()). * **BEGIN(etat)** positionne l’automate dans la condition de départ état. Cet état doit avoir été défini dans la première section grâce à %Start etat ou à %x etat. BEGIN(0) permet de revenir à l’état normal. * **int main()** par défaut, la librairie de lex (libl.a) ou de flex (libfl.a) définissent une fonction principale qui appelle yylex() jusqu’à ce que celle-ci retourne 0. |
| --- |

Ambiguïté de correspondance (Règle de la plus longue correspondance) : Si un préfixe (début de chaîne) correspond à plusieurs expressions régulières possibles, lex choisira l’expression régulière correspondant à la plus longue extension.

Règle du premier trouvé : Si la longueur de correspondance est égale pour plusieurs règles, alors c’est la première dans la liste qui est déclenchée.

En fait, les opérateurs contextuels de suffixe ($, /) sont consommés après le lexème et c’est ce mot qui doit être considéré comme le plus long possible. Ensuite, le suffixe sera rejeté dans yyin

**Définitions dans un fichier :**

Abréviation de modèle certains facteurs de modèles revenant fréquemment dans les règles, on peut en définir des alias selon la syntaxe suivante : nom Alias séparateur(s) modèle. Par ex :

**chiffre ([0-9])** ou bien l’alias **minuscule ([a-z])** ou encore **exposant ([DEde][-+]?{chiffre}+).**

| **Définition** - Start conditions : permet de conditionner la reconnaissance de certaines expressions régulières selon l’état dans lequel l’analyseur se trouve. Par exemple, %Start state1 state2 state3 définit trois états. Ceux-ci pourront être utilisés en préfixe  des expressions régulières : <state2>[a-z]+ {BEGIN(state3);}. |
| --- |

**Troisième section :**

* Cette section permet d’écrire des fonctions C utilisées dans les parties droites des règles.
* On peut également redéfinir les fonctions main(), yywrap(), input(), unput(char), ... afin de surcharger leur version flex.
* Ces fonctions peuvent également être redéfinies dans un fichier inclus.
* Enfin, on peut utiliser des fichiers objets externes lors de l’édition de liens à condition d’avoir inclus leurs en-têtes dans la première section.

**Les options de la commande flex :**

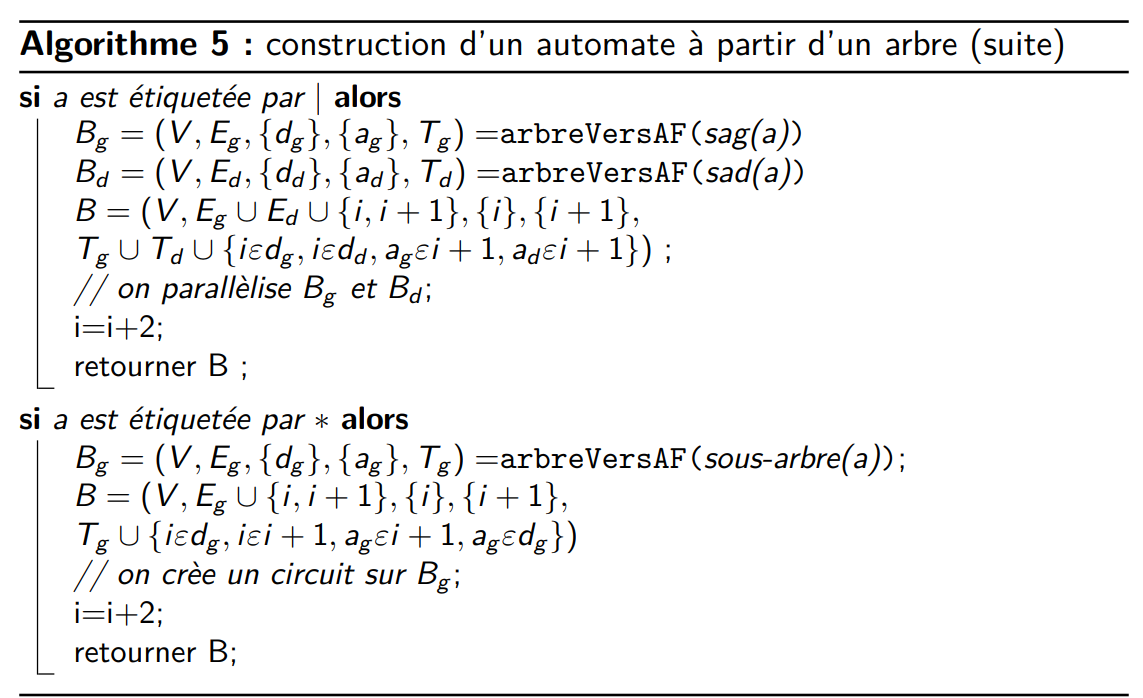
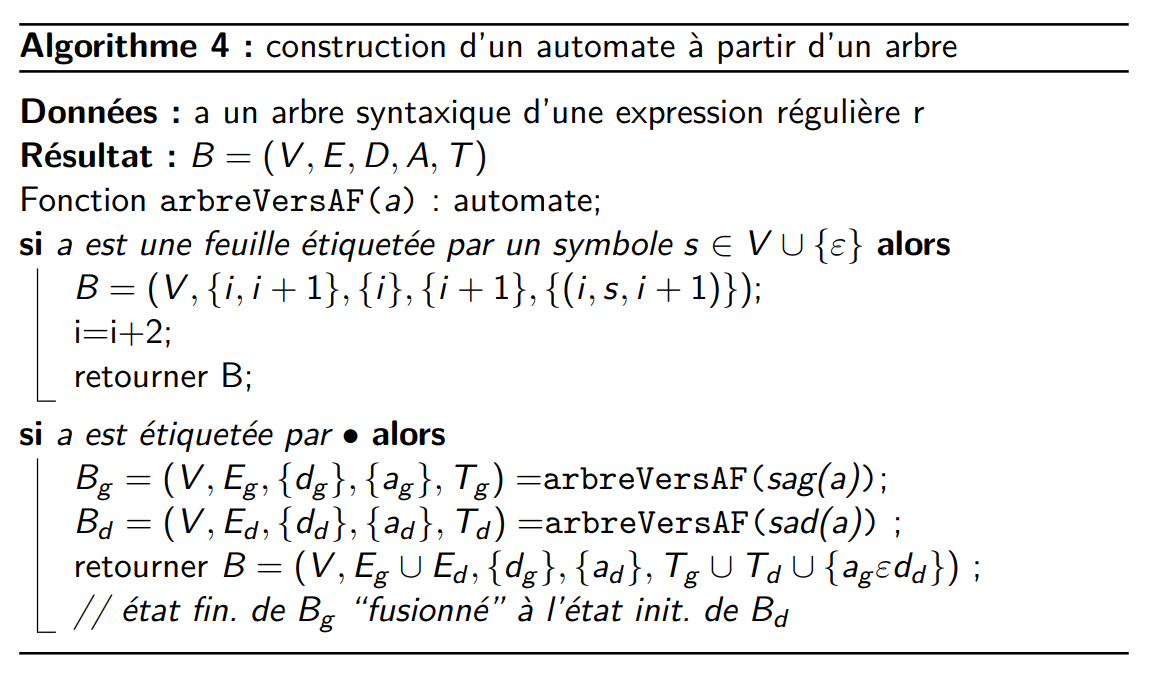
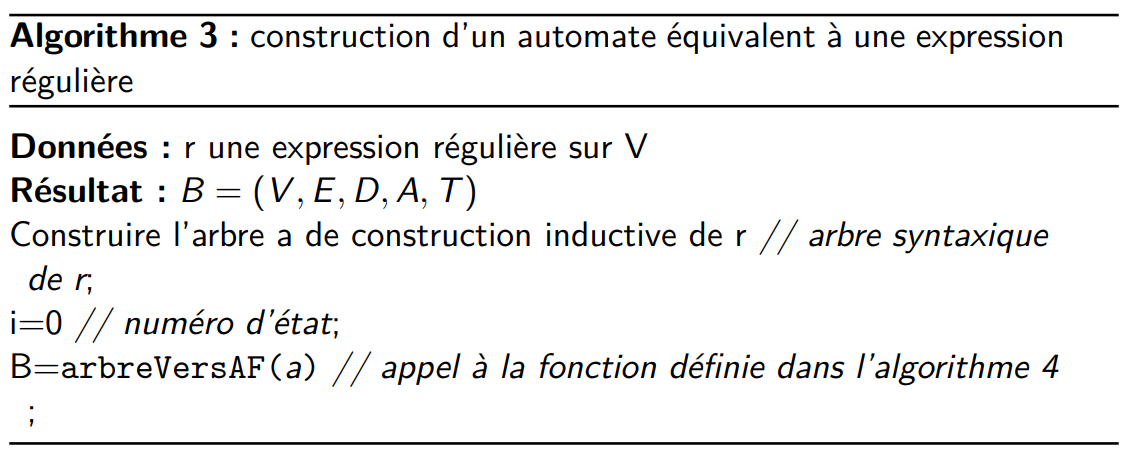
| flex -d | débogue un source flex en affichant lors de l’exécution la règle reconnue (ligne) et le lexème ; flex -T trace l’automate construit en donnant |
| --- | --- |
| flex -T | trace l’automate construit en donnant : L’AFN (nfa), l’AFD (dfa), et les classes de caractères définies |
| flex -v | (verbose) donne des informations statistiques sur l’automate généré ; |
| flex -s | supprime la règle par défaut qui consiste à envoyer sur la  sortie standard tout caractère non reconnu |

**Liaison avec un analyseur syntaxique :**

Lorsqu'il est utilisé avec un analyseur syntaxique généré par yacc ou bison, c’est la fonction d’analyse syntaxique yyparse() qui appelle itérativement yylex() pour obtenir les jetons correspondants au fichier analysé. La fonction principale int main() appelle alors yyparse() et non plus yylex(). Une ou plusieurs variables globales, yylval par exemple, peuvent être alors partagées par les 2 fonctions yylex() et yyparse().

**Traduction des expressions régulières et algorithme de Thompson :**

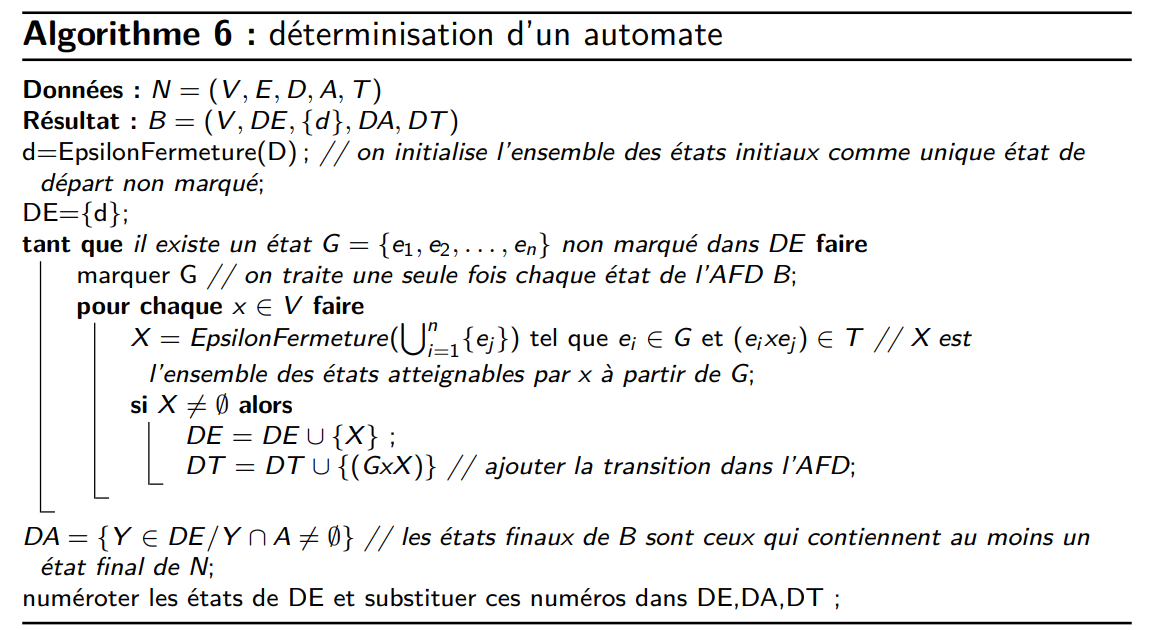
construction de “Thompson” qui admet des AFN possédant des ε-transitions mais ayant un unique état initial et un unique état final. Le résultat est un AFN. Le principe revient à associer récursivement un automate à chaque nœud de l’arbre syntaxique de l’expression régulière.



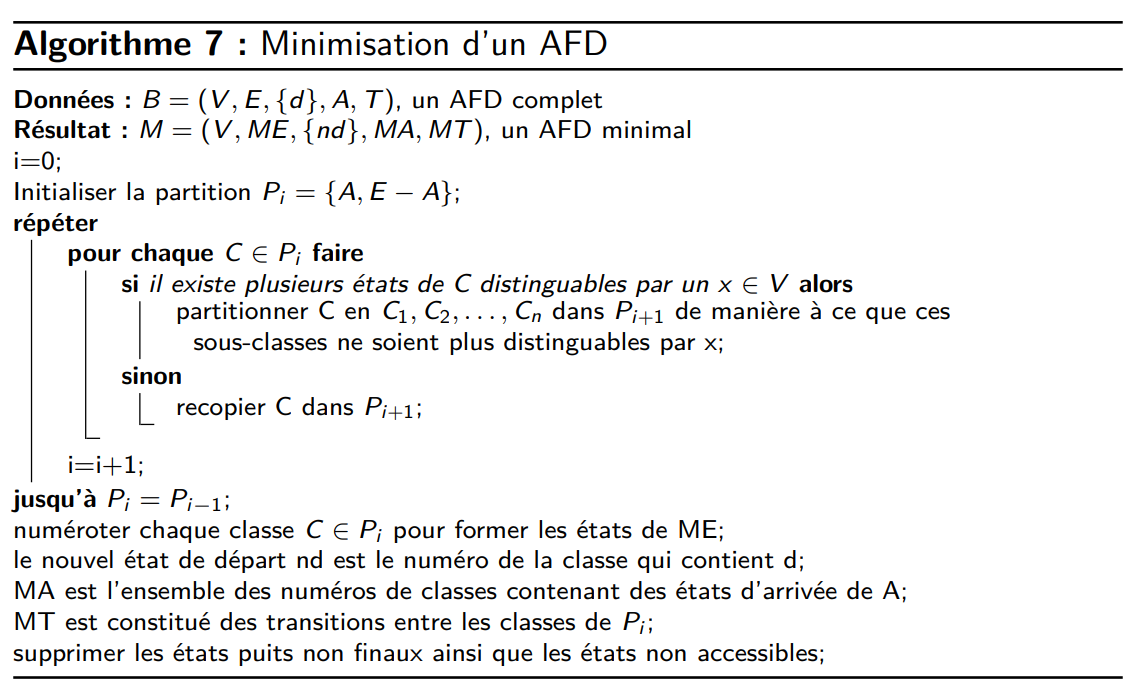
**Propriétés de l’algorithme de Thompson :**

* L’AF construit a au plus deux fois plus d’états que |r|. Il a un état initial et un état final.
* Chaque état (non final) possède soit 1 ou 2 ε-transitions sortantes, soit une transition sortante étiquetée par un symbole de V
* Chaque état (non initial) possède, soit 1 ou 2 ε-transitions entrantes, soit une transition entrante étiquetée par un symbole de V.
* L’état final n’a pas de transition sortante, l’état initial n’a pas de transition entrante.

**Déterminisation d’un automate :**

Remarques : 

* A tout chemin menant d’un état initial à un état final de N, donc à tout mot de L(N), correspond un chemin de d à un état final dans D.
* De plus, pour un chemin menant à un état final, l’état {. . . en+1 . . .} est final (Voir dans l’algorithme : DA = {Y ∈ DE/Y ∩ A 6= ∅}).
* Remarquons que cette détermination permet de supprimer tous les chemins inaccessibles

**Minimisation** :

* Rappelons que la forme canonique d’un langage régulier est son AFD minimal.
* l’algorithme de minimisation d’un AFD B = (V, E, {d}, A,T) suppose en entrée un AFD complet en ajoutant si nécessaire un état puits
* On va construire incrémentalement une suite de partitions Pi, composées de classes d’états
* On dit que 2 états i, j d’une même classe C sont distinguables par un symbole x ∈ V ssi la reconnaissance de x n’aboutit pas pour ces deux états à la même classe de la partition courante
* On va partitionner les états de l’automate en classes d’états distinguables les unes par rapport aux autres puis ces classes représenteront les états du nouvel AFD Minimal M.

**Exemple :**

B = ( V = {a, b}, E = [1, 6], D = {1}, A = {3, 4, 5}, T = {1a2, 1b3, 2a2, 2b3, 3a4, 3b6, 4a5, 4b6, 5a5, 5b6, 6a6, 6b6})

* On obtient la partition initiale : P0 = {{3, 4, 5}, {1, 2, 6}} La classe {3, 4, 5} n’est pas distinguable ni par a (classe {3, 4, 5}), ni par b (classe {1, 2, 6})
* Par contre, la classe {1, 2, 6} se distingue sur b, par conséquent :
* P1 = {{3, 4, 5}, {1, 2}, {6}} = P2

Il ne reste plus qu’à supprimer la classe {6} qui est un puits non final pour obtenir l’AFD minimal : M = ({a, b}, {12, 345}, {12}, {345}, {12a12, 12b345, 345a345})

Remarquons qu’un état d’arrivée de M ne contient que des états d’arrivée de B à cause de la partition initiale.

**Analyse syntaxique :**

L’analyse syntaxique du programme source doit vérifier que celui-ci est bien un mot du langage de programmation. Pour cela, la grammaire du langage est utilisée. Cette grammaire G = (VT, VN, R, S) est algébrique (insensible au contexte). Toutes les règles de R sont donc de la forme : X → α avec X ∈ VN et α ∈ (VT ∪ VN)∗. Il existe un unique arbre de dérivation dont la frontière soit le programme.

Cette analyse peut se faire selon deux approches :

* l’analyse syntaxique descendante consiste à partir de l’axiome qui constitue la racine de l’arbre de dérivation (ou arbre syntaxique) l’arbre de dérivation est ainsi construit (ou pas) depuis la racine S vers les feuilles.
* l’analyse syntaxique ascendante consiste, au contraire, à partir du programme et à remonter vers l’axiome S l’arbre de dérivation est alors construit (ou pas) depuis les feuilles vers la racine S

Un arbre syntaxique représentant le programme. Celui-ci est soit un arbre de dérivation (arbre complet), soit un arbre abstrait (Abstract Syntax Tree) qui est un arbre simplifié. Cet arbre servira ensuite pour l’analyse sémantique puis la synthèse de la cible ou l’évaluation.

| **Analyse descendante récursive** :  Méthode de programmation qui associe une fonction, souvent récursive, à chaque symbole non terminal de la grammaire. Ces fonctions s’appellent suite à la reconnaissance de certains jetons du flot d’entrée correspondant aux début des parties droites des règles de production. Ces jetons permettent donc de prédire la règle de production à choisir. propriété fondamentale de ces grammaires : non récursivité à gauche.  La récursivité à droite étant permise, il est toujours possible de transformer une grammaire récursive à gauche en une grammaire équivalente non récursive à gauche. Le nombre de symboles terminaux nécessaires à la prédiction de la règle de production à choisir est une caractéristique des analyses descendantes prédictives. Si ce nombre est 0, on choisit une production quelconque et on tente la descente.  Si celle-ci échoue : backtracking. Le backtracking étant coûteux du point de vue de l’efficacité, on utilise toujours au moins un symbole (jeton) de prédiction (prévision). ce jeton doit être lu avant d’entrer dans une fonction afin de permettre le retour sans effet dans le cas d’une production epsilonesque. |
| --- |

Soit la grammaire d’expressions arithmétiques intuitive suivante :

GE = ({0, 1, . . . , 9, +, ∗,(,)}, {E}, R, E) avec les règles de R suivantes :

E → E + E|E ∗ E|(E)|0|1| . . . |9

Cette grammaire GE étant ambigüe, on écrit une grammaire équivalente non ambigüe selon le schéma Expression Terme Facteur (ou ETF) :

* une expression est quelconque, par exemple 1+2\*3+4 ;
* un terme est un élément d’une somme : dans l’exemple précédent, 1, 2\*3 et 4 sont trois termes ;
* un facteur est un élément d’un produit : dans l’exemple précédent, 2 et 3 sont des facteurs du produit 2\*3.

Analdesc permet cette vérification : l’analyse lexicale est triviale car chaque lexème du langage est constitué d’un seul caractère (getchar()). on utilisera deux variables globales jeton et numcar pour conserver le jeton courant et la position dans la ligne. Il contient les macros suivantes :

* lire le jeton suivant (AVANCER())
* comparer le jeton courant avec le jeton attendu puis avancer (TEST\_AVANCE())
* gérer les erreurs de syntaxe (ERREUR\_SYNTAXE())

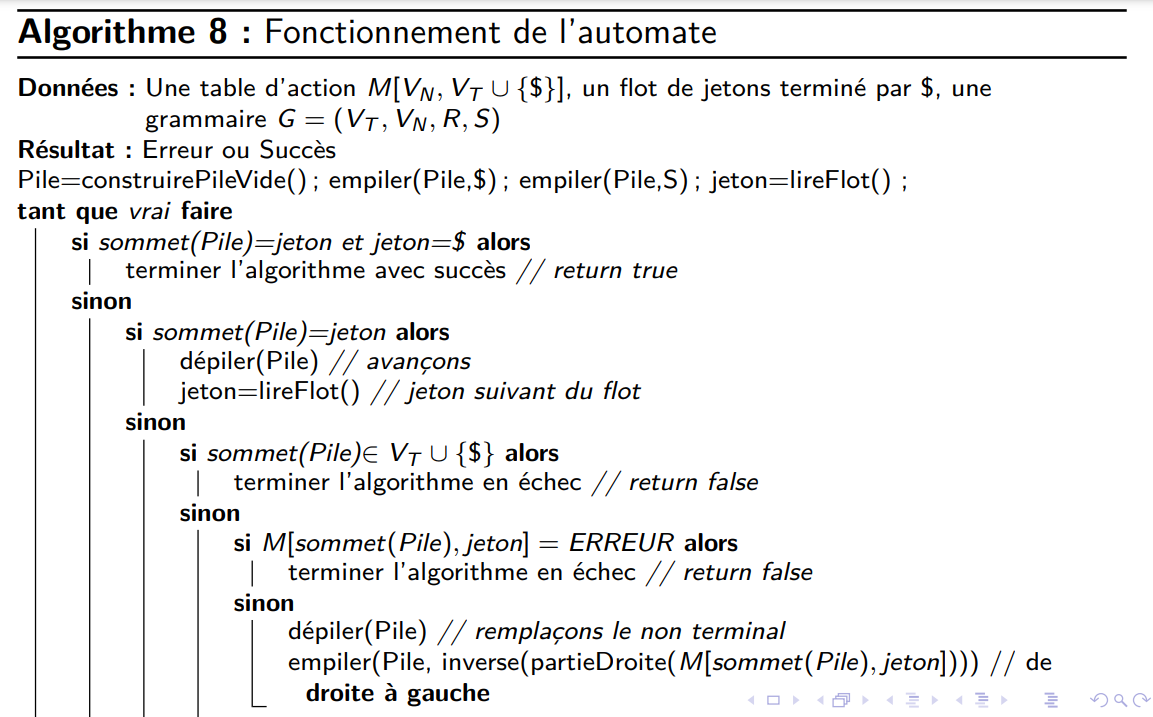
| **Définition** : Automate à pile : un automate à pile est une machine lisant itérativement des symboles terminaux (jetons) depuis le flot d’entrée, gérant une pile de symboles, et exécutant des actions en fonction d’une table d’analyse ou table d'action. |
| --- |

Le flot d’entrée est constitué d’une suite de jetons terminée par un symbole spécial de fin représenté par **$** (jeton 0 retourné par yylex()).la pile est toujours initialisée avec le symbole spécial $ puis est manipulée par des empilements et dépilements dépendant de la table d’actions. Ainsi, en fonction du symbole de sommet de pile et du jeton courant, la table indique l’action à réaliser.

| **Définition** - Table d’action : la table d’actions est une table à 2 dimensions indicées par les non terminaux d’une part, et les symboles terminaux (jetons du flot) et $ d’autre part |
| --- |

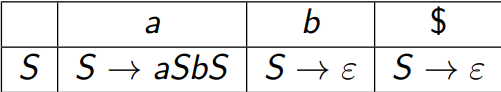
**Fonctionnement de l’automate à pile en analyse descendante :**

Soit la grammaire G = (VT , VN, R, S), chaque case de la table M[VN, VT ∪ {$}] contient :

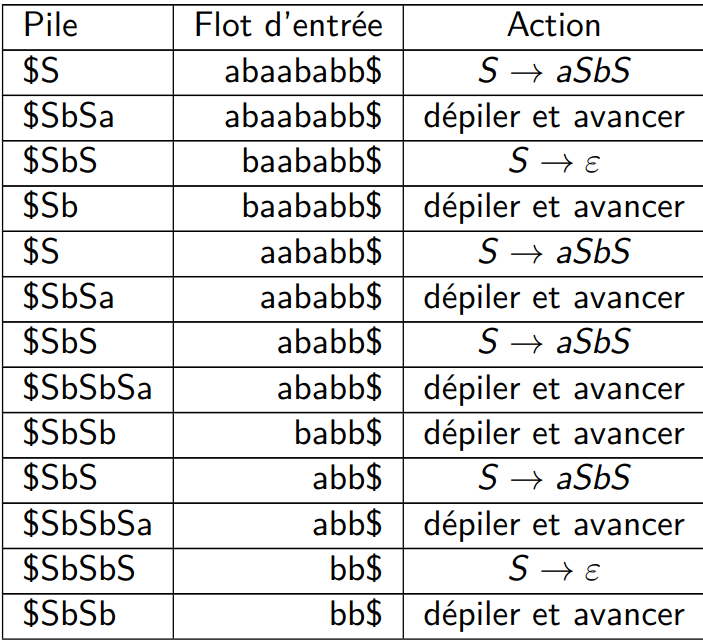
* soit une règle de production
* soit l’action ERREUR

A tout moment, l’analyse du flot d’entrée consiste à regarder la règle de production correspondant au sommet de pile et au jeton d’entrée. Puis, selon les cas, l’automate soit :

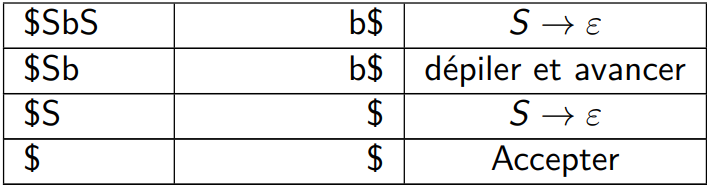
* s'arrête en générant une erreur de syntaxe,
* avance sur le flot et dépile un jeton (concordance),
* empile à l’envers la partie droite de la règle,
* termine en indiquant la réussite de l’analyse.

Exemple : Une grammaire de Dyck à un couple de parenthèses : GD = ({a, b}, {S}, R, S) avec les règles de R suivantes : **S → aSbS|ε**

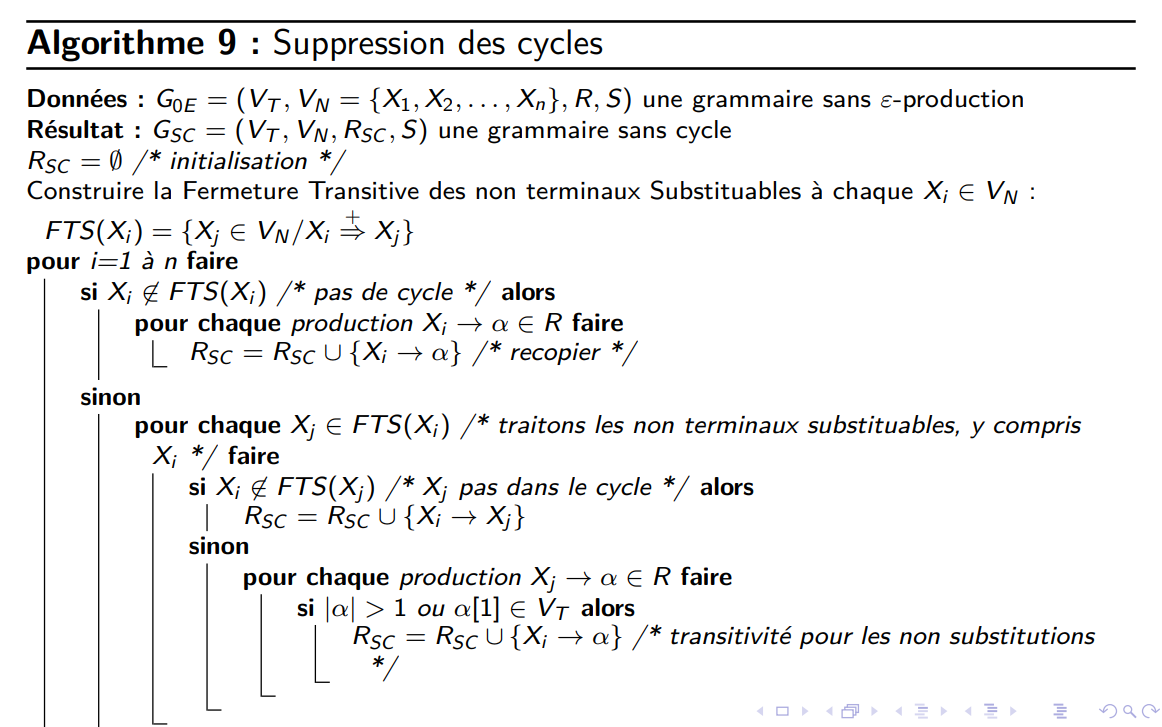
On obtient la table d’analyse suivante (voir algorithme 16) :



Etudions le fonctionnement de l’automate, c’est-à-dire de sa pile, sur le mot d’entrée abaababb$ :



| **Suppression des ε-productions :**  Les symboles non terminaux effaçables, c’est-à-dire pouvant dériver en ε, sont détectés de la manière suivante. Un symbole non terminal effaçable :   * soit dérive directement en ε, * soit dérive en un mot constitué exclusivement de symboles non terminaux effaçables.   Soit G = (VT , VN, R, S), soit Ei une suite d’ensembles Effaçables de symboles non terminaux définie comme suit :   * E1 = {X ∈ VN/(X → ε) ∈ R} * Ei+1 = Ei ∪ {X ∈ VN/(X → α) ∈ R et α ∈ Ei∗}   On prouve que les ensembles Ei ne contiennent que des symboles non terminaux effaçables, c’est à dire dérivant en ε  On prouve également que la suite Ei converge et est donc constante  au-delà d’un certain rang n :   * ∃n ∈ N, En = En+k , ∀k ∈ N * Par conséquent, ∀X ∈ VN, X∗⇒ ε si et seulement si X ∈ En. * Il reste à construire une grammaire GSE ne contenant (presque) plus d’ε-production et équivalente à G * Il peut rester une ε-production dans le cas où le langage de la grammaire contient le mot vide. . . * Soit G0E = (VT , VN, R1, S) avec un ensemble de règles défini comme suit : * R1 = {X → α tel que α 6= ε et ∃X → β ∈ R tel que α s’obtient à partir de β en supprimant un nombre quelconque (k ∈ [0, |β|[) d’occurrences d’éléments effaçables (de En)} * On prouve que L(G0E ) = L(G) − {ε}. Si S est un symbole effaçable de G, S ∈ En, on obtient GSE en ajoutant un nouvel axiome S1 et deux nouvelles règles : * GSE = (VT , VN ∪ {S1}, R1 ∪ {S1 → ε|S}, S1) * Sinon, S !∈ En, on a GSE = G0E.   **Exemple :**  Soit la grammaire G = ({a, b}, {S, X, Y }, R, S) avec les règles de R suivantes :  S → aX|Y |XX  X → ε|b|XX  Y → aXb  On calcule les ensembles d’effaçables : E1 = {X}, E2 = {X, S}, E3 = {X, S}. On obtient donc un nouvel ensemble de règles R1 :  S → aX|a|Y |XX|X  X → b|XX|X  Y → aXb|ab  Pour finir, voici la grammaire équivalente à G et ne contenant qu’une ε-production :  GSE = (VT , VN ∪ {S1}, R1 ∪ {S1 → ε|S}, S1). |
| --- |

**Suppression des cycles :** 

* On suppose une grammaire sans ε-production.
* L’algorithme 9 supprime les cycles de dérivation : X+⇒ X.
* Une production est appelée substitution de non terminal ou plus simplement substitution lorsqu’elle est de la forme : X → Y .
* Seules les substitutions engendrant des cycles doivent être supprimées.
* Dans l’algorithme 9, on calcule la Fermeture Transitive des non terminaux Substituables à chaque symbole non terminal.
* Ce calcul partitionne VN en classes d’équivalence correspondant aux cycles de non terminaux substituables.
* Puis on filtre les productions selon l’appartenance de leur partie gauche à un cycle.

Soit la grammaire G = ({a, b, c, d}, {X1, X2, X3}, R, X1) avec les règles de R suivantes :

X1 → X2|a

X2 → X1|X2|X3|b

X3 → bX1|X2a

On calcule les fermetures transitives des substituables : FTS(X1) = {X1, X2, X3}, FTS(X2) = {X1, X2, X3}, FTS(X3) = ∅. On obtient donc un nouvel ensemble de règles sans cycle RSC :

X1 → a|b|X3

X2 → a|b|X3

X3 → bX1|X2a

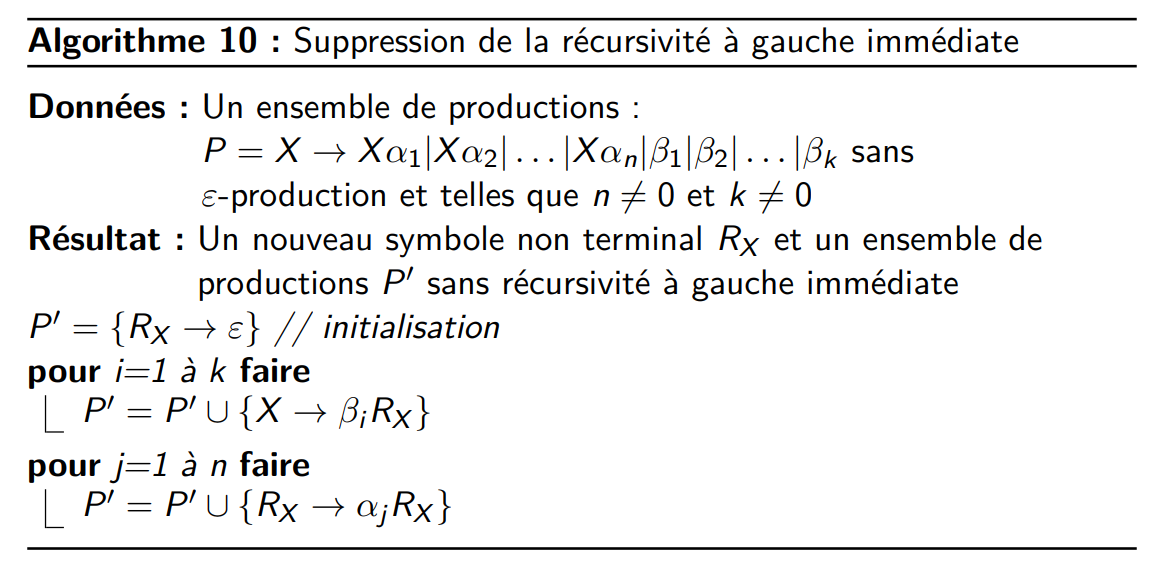
Remarquons que X1 et X2 peuvent être remplacés par X1 qui les représentent tous deux. Ce qui donne :

X1 → a|b|X3

X3 → bX1|X1a

**Suppression de la récursivité à gauche immédiate :**

* Une récursivité à gauche immédiate d’un symbole non terminal X se matérialise par au moins une règle de production X → Xα
* La suppression de cette récursivité à gauche immédiate nécessite de transformer l’ensemble des règles de production ayant X comme partie gauche (les X-productions)
* L’algorithme 10 réalise cette transformation
* Remarquons que l’appel de cet algorithme nécessite d’avoir au moins une récursivité à gauche immédiate (n 6= 0) et au moins une autre production (k 6= 0)
* Cette dernière condition est indispensable dans une grammaire sans ε-production
* Sinon, le non terminal X ne peut dériver en un mot terminal !



**Remarques :**

* L’algorithme 10 crée un nouveau symbole RX (Reste de X), pour remplacer la récursivité à gauche par une récursivité à droite sur RX
* Remarquons que RX possède une ε-production donc est effaçable
* La correction de l’algorithme, c’est-à-dire l’équivalence des deux ensembles de productions P et P’, se démontre par une double récurrence sur i et j

**Exemple** : Soit la grammaire d’expressions arithmétiques

GE = ({0, 1, . . . , 9, +, ∗,(,)}, {E}, P, E) avec les règles de P suivantes :

E → E + E|E ∗ E|(E)|0|1| . . . |9

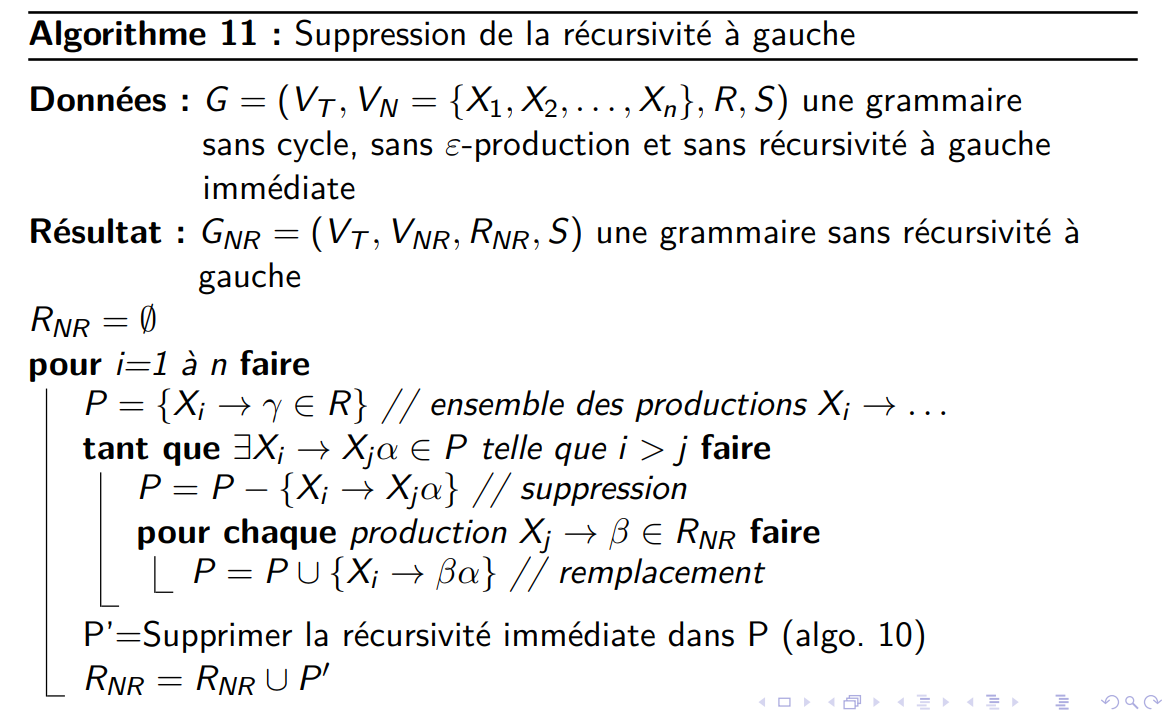
Après application de l’algorithme 10, on obtient la grammaire suivante :

GENRI = ({0, 1, . . . , 9, +, ∗,(,)}, {E, RE }, P’, E) avec les règles de P’ suivantes :

E → (E)RE |0RE |1RE | . . . |9RE

RE → ε| + ERE | ∗ ERE

Remarquons que GENRI n’est plus récursive à gauche, mais elle reste ambiguë.



* Dans certains cas, la suppression de la récursivité à gauche immédiate ne suffit pas car il peut subsister des récursivités plus complexes
* dans les productions X1 → X2a|a, X2 → X1b|b il n’y pas de récursivité à gauche immédiate mais il y a de la récursivité à gauche !
* L’algorithme 11 s’applique à une grammaire sans cycle, sans ε-production et sans récursivité à gauche immédiate. Il produit une grammaire sans récursivité à gauche, c’est-à-dire sans dérivation de la forme X+⇒ Xα.

**Exemple** : Soit la grammaire G = ({a, b, d}, {X1, X2, X3}, P, X1) avec les règles de P suivantes :

X1 → X2a|d

X2 → X3a|X1b

X3 → X1a

Après application de l’algorithme 11, on obtient la grammaire suivante G’ = ({a, b, d}, {X1, X2, R2, X3, R3}, P’, X1) avec les règles de P0 suivantes :

X1 → X2a|d

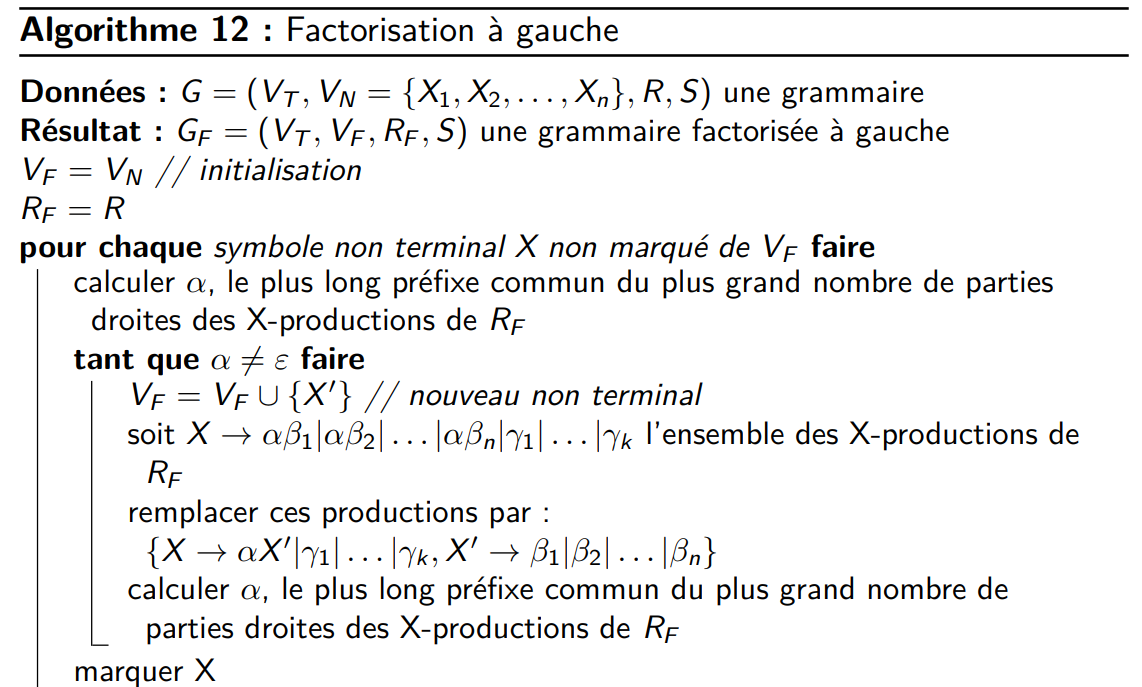
X2 → X3aR2|dbR2

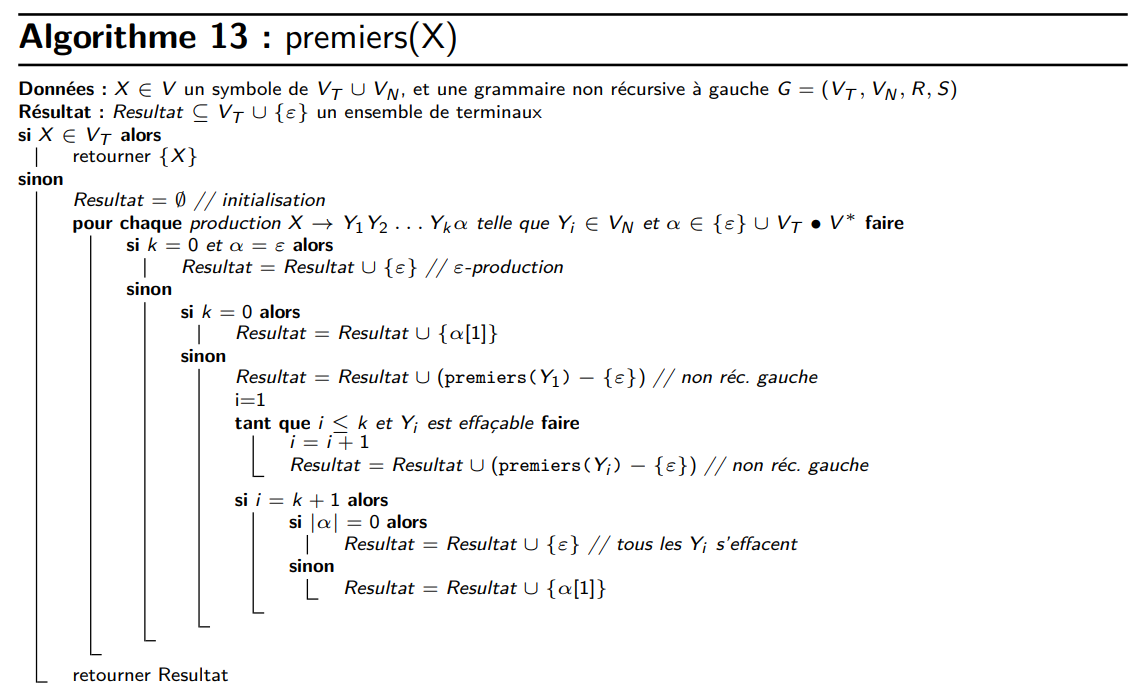
R2 → ε|abR2

X3 → dbR2aaR3|daR3

R3 → ε|aR2aaR3

**Factorisation à gauche :**

* Si plusieurs parties droites de X-productions ont même préfixe, la prédiction de la règle à choisir est retardée jusqu’à ce qu’un jeton permette de déterminer la “bonne” règle
* Il faut donc pouvoir lire plusieurs jetons en avance !
* La factorisation des parties droites est destinée à réduire à 1 ce nombre de jetons de prévision
* les grammaires ainsi formées seront qualifiée de LL(1) (Left to right scanning of the input, Leftmost derivation, 1 look-ahead symbol)

**La fonction premier() :** 

* La fonction premiers est nécessaire à la construction de la table d’analyse qu’utilise l’automate à pile
* Elle retourne un ensemble de terminaux (jetons)
* premiers suppose une grammaire non récursive à gauche mais pouvant admettre des ε-productions
* La fonction premiers(α) retourne l’ensemble des terminaux qui débute un mot dérivant de α
* Si α est effaçable alors ε fait partie de ses premiers
* Pour calculer premiers(α), il faut commencer par calculer premiers(X), quel que soit X un symbole de V
* L’algorithme 13 réalise cette fonction.

**Remarques :**

* Pour les non terminaux, il consiste à accumuler les premiers(Yi) tant que Yi−1 est effaçable
* ε n’est ajouté que dans le cas ou une partie droite de production est entièrement effaçable
* cet algorithme ne peut être utilisé sur une grammaire récursive à gauche (appel récursif infini)

**Exemple** : Soit la grammaire non récursive à gauche GENR = ({0, 1, . . . , 9, +, ∗,(,)}, {E, R,T, S, F}, X, E) avec les règles de X suivantes :

E → TR

R → +TR|ε

T → FS

S → ∗FS|ε

F → (E)|0|1| . . . |9

On obtient par l’application de l’algorithme 13 :

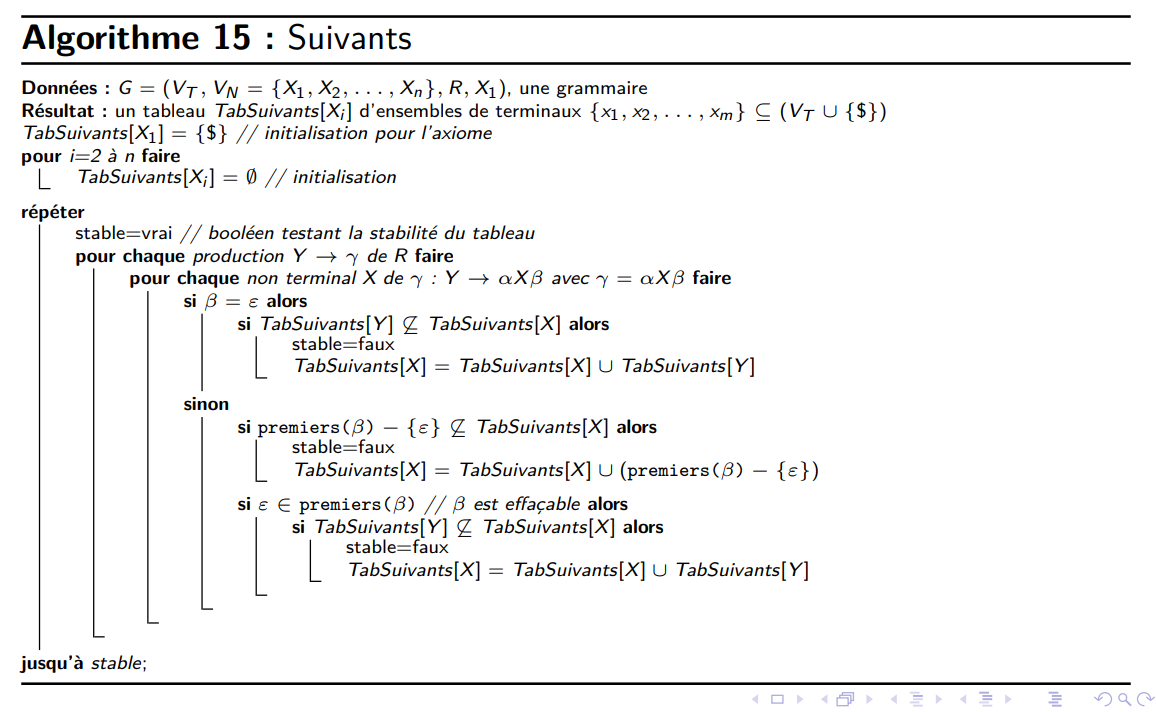
premiers(F) = {(, 0, 1, . . . , 9}

premiers(S) = {∗, ε}

premiers(T) = premiers(F)

premiers(R) = {+, ε}

premiers(E) = premiers(F)



L’algorithme 15 est nécessaire à la construction de la table d’analyse qu’utilise l’automate à pile Il utilise une grammaire G et calcule un tableau d’ensembles de terminaux, et éventuellement $ le symbole de fin d’entrée.

**Exemple** : Soit la grammaire non récursive à gauche GENR de l’exemple de la page 162. On obtient par l’application de l’algorithme 15 :

TabSuivants[E] = {$,)}

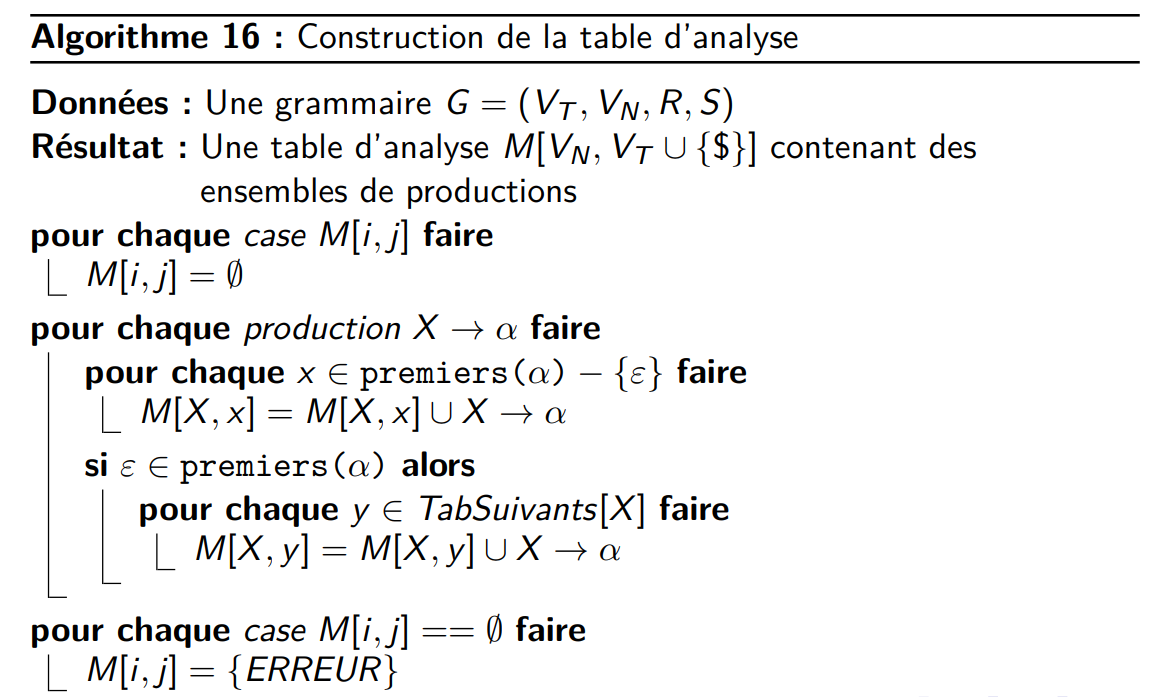
TabSuivants[T] = {+, $,)}

TabSuivants[R] = {$,)}

TabSuivants[F] = {∗, +, $,)}

TabSuivants[S] = {+, $,)}

**Construction de la table d’analyse :**

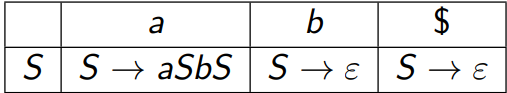
L’algorithme 16 réalise la construction de la table d’analyse qu’utilise l’automate à pile. Dans cette table, l’existence de plus d’une production dans une case est appelée un conflit et signifie que l’automate à pile à un choix à réaliser ! Ceci n’est pas envisageable pour des raisons d’efficacité (backtrack) il sera alors nécessaire de transformer la grammaire.

**Exemple** :

Reprenons l’exemple de la grammaire de Dyck à un couple de parenthèses a, b

* Soit la grammaire GD = ({a, b}, {S}, R = {S → aSbS|ε}, S)
* La première règle S → aSbS ne pose aucun problème car La premiers(aSbS) = a donc M[S, a] = S → aSbS
* Quant à la seconde production S → ε, elle génère le calcul de TabSuivants[S]={b, $}

On obtient donc la table d’analyse suivante :



Reprenons une grammaire non récursive à gauche plus complexe et voyons la table d’analyse générée GENR = ({0, 1, . . . , 9, +, ∗,(,)}, {E, R,T, S, F}, X, E) avec les règles de X

suivantes :

E → TR

R → +TR|ε

T → FS

S → ∗FS|ε

F → (E)|0|1| . . . |9

Il nous faut rappeler les premiers() des non terminaux débutant des

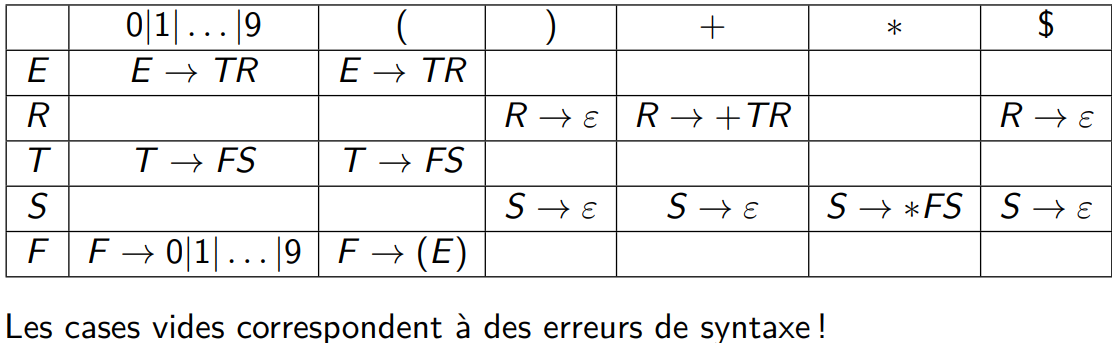
parties droites :

* premiers(F) = {(, 0, 1, . . . , 9}
* premiers(T) = premiers(F)

Il nous faut également rappeler les suivants des non terminaux effaçables :

* TabSuivants[R] = {$,)}
* TabSuivants[S] = {+, $,)}

On obtient finalement par l’application de l’algorithme 16, la table suivante :



Voir contre exemple page 174 du CM.

**Grammaire LL(1) :**

| **Définition** : Une grammaire dont la table d’analyse peut être calculée et dont toutes les  entrées ont une unique production ou bien ERREUR, est appelée LL(1). Left to Right scanning of the input, Leftmost derivation, 1 symbole de prévision. |
| --- |

| Théorème : Aucune grammaire ambiguë et aucune grammaire récursive à gauche n’est  LL(1). |
| --- |

| Théorème : Une grammaire G est LL(1) si et seulement si les conditions suivantes sont  respectées. Quelle que soit X → α|β, deux productions de G :   * il n’existe pas deux dérivations de α et β ayant un préfixe commun terminal ; * une partie droite seulement, α ou bien β, peut s’effacer ; * si α peut s’effacer, alors β ne dérive pas en un mot ayant un préfixe commun terminal avec suivants(X). |
| --- |

**Conclusion analyse descendante :**

* Examinons les grammaires qui ne sont pas LL(1) :
* Toutes les grammaires ambiguës ne sont pas LL(1)
* Certaines grammaires non ambiguës ne sont pas LL(1)
* Par exemple, G2 = ({a, b}, {S, A}, {S → Ab|aa, A → a}, S) est une grammaire simple produisant 2 mots aa et ab et n’est pas LL(1). En effet, sur la lecture du premier a, on ne peut pas déterminer quelle production de S utiliser.
* Cependant, on peut parfois utiliser un automate à pile en analyse descendante pour reconnaître le langage généré par une grammaire non LL(1) (exemple de la grammaire if then else qui génère un conflit). On peut déterminer cette table en réussissant à reconnaître le même langage. Malheureusement, ce problème du choix est indécidable et nécessite donc une réflexion ad hoc.
* Dans l’exemple de G2, le choix de l’une ou de l’autre des productions de S à privilégier aboutit à un langage reconnu réduit de moitié !
* Aussi, les actions sémantiques qu’il faut associer à ces règles syntaxiques deviennent difficiles à mettre en oeuvre
* L’analyse syntaxique ascendante va nous permettre de conserver des grammaires récursives à gauche et/ou à droite plus intuitives

**Bison et Yacc :**

* Yacc (“Yet Another Compiler Compiler”) est un outil d’analyse syntaxique permettant d’écrire des grammaires algébriques LALR(1) assez générales (“Look Ahead Left to right scanning of the input, Rightmost derivation in reverse, 1 look-ahead token”).
* Il génère un analyseur syntaxique ascendant utilisant un automate à pile
* Associés à chaque règle de grammaire, des actions peuvent être associées

**Un exemple :**

* Au cœur du source C y.tab.c généré par bison, la fonction C : int yyparse() d’analyse syntaxique permet de retourner la valeur 1 en cas d’erreur syntaxique, 0 sinon
* La fonction principale : int main() appelle yyparse() qui va appeler yylex() itérativement au fur et à mesure de la reconnaissance des règles de grammaires
* En cas d’erreur de syntaxe, yyparse() fait appel à yyerror(char\*) pour informer l’utilisateur puis yyparse() retourne 1.
* L’option -y de bison permet de générer un fichier nommé y.tab.c, comme en yacc
* Sans cette option, le fichier généré se nommerait veriflog.tab.c.

**Un source bison comprend 3 parties séquentielles :**

* une partie déclaration contenant des déclarations C contenues entre %{ et %}, et des déclarations spécifiques à bison.
* Délimitée par %% au début, une partie constituées de règles de grammaire et des actions associées à la reconnaissance de chaque règle. C’est la partie centrale du source bison qui définit l’analyseur syntaxique.
* Délimitée par %% au début, une partie de fonctions C définies par l’utilisateur. Dans le cas de Bison, on doit définir au moins trois fonctions : le main(), yyerror() et yylex()
* Remarquons que ces fonctions peuvent être définies dans un autre fichier qui sera lié après compilation
* Dans le cas de Yacc, une librairie liby.a contient des définitions par défaut de ces trois fonctions.

**Les règles de grammaire bison :**

* Une règle bison se présente de la façon suivante : un symbole non terminal, le caractère “ :”, une séquence de symboles (terminaux (jetons) ou non terminaux) et de blocs d’actions {...}, terminé par un” ;”
* L’espace, la tabulation et le retour à la ligne ne sont pris en compte que comme séparateurs
* La règle doit commencer en début de ligne et terminer par un “ ;”.

**Les symboles terminaux (jetons) et non terminaux :**

Les symboles terminaux ou jetons sont représentés par un entier (int) retourné par la fonction d’analyse lexicale yylex(). Les jetons peuvent être :

**non nommés** comme ’&’, ’1’ dans l’exemple précédent. En fait, dans cet exemple, tous les jetons étaient non nommés.

**ou bien nommés**  Dans ce cas, yylex() et yyparse() doivent partager une définition (#define) commune de ces jetons. La manière la plus simple consiste à :

1. Les déclarer, dans la première partie du source bison à l’aide du déclarateur bison : %token NAME. Par convention, les noms de jeton sont en majuscules
2. Générer un fichier y.tab.h contenant les #define correspondant grâce à l’option -d du compilateur bison
3. Inclure ce fichier dans la partie définition du source flex.

* Les caractères ascii ont un numéro de jeton égal à leur code ascii !
* Enfin, un jeton spécial error est réservé pour la gestion des erreurs.
* Les symboles non terminaux sont conventionnellement écrits en minuscules (expr, statement, . . . ).

**Partie droite de règle :**

Les différentes productions associées au même non-terminal seront séparées par une barre verticale “|”. Une partie droite peut être vide afin d’indiquer une epsilon-production. Par exemple :

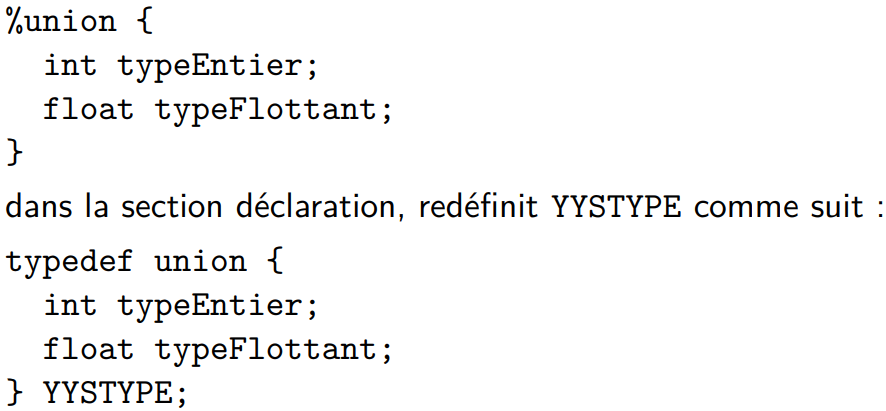
list : // epsilon-production

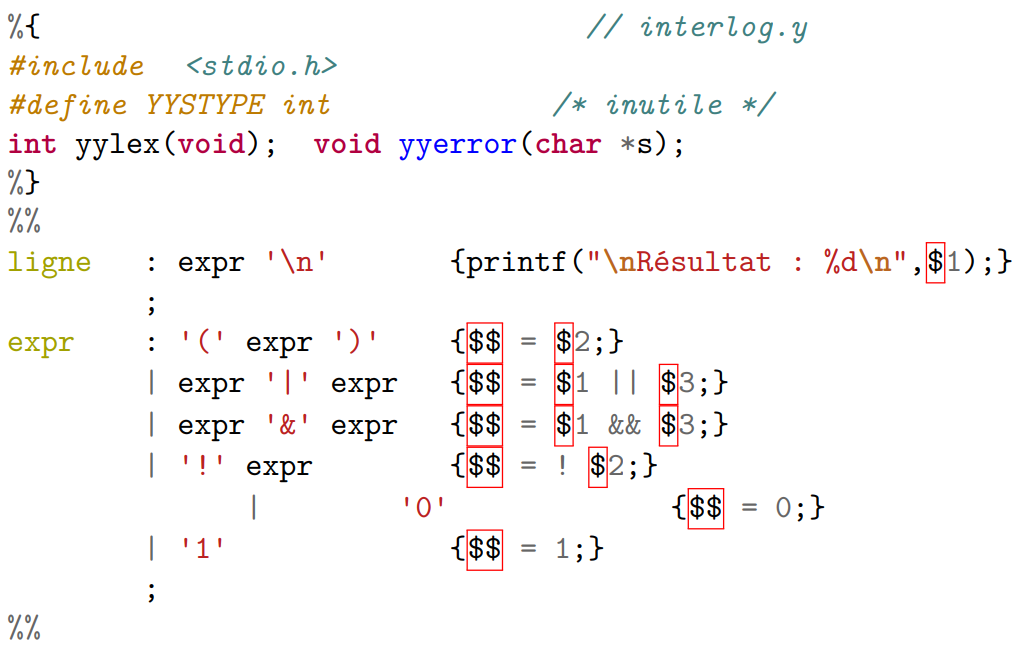
| list ',' stat

;

**Valeur sémantique** :

Associée à chaque symbole, terminal ou non, une valeur sémantique (attributs des grammaires attribuées) est définie automatiquement par bison. Le type YYSTYPE (YY Semantic TYPE) par défaut de cet attribut est entier (int) mais peut être défini de deux façons :

* si l’on a besoin que d’un seul type sémantique pour tous les symboles de la grammaire, il suffit de définir YYSTYPE par une macro dans les déclarations C : #define YYSTYPE double ;
* attention à répéter cette macro également dans le source flex avant l’inclusion de y.tab.h sinon lex utilisera le type par défaut int
* Si l'on a besoin de plusieurs types sémantiques pour différents symboles, par exemple int et float, on utilisera la déclaration bison union. Par exemple :
* La variable globale yylval est l’attribut que yylex() peut affecter aux jetons



**Les actions** :

* N’importe quelle instruction C peut apparaître dans un bloc d’actions
* De plus, bison admet des actions spécifiques permettant d’utiliser les attributs
* L’attribut associé à la partie gauche de la règle de production courante est nommé $$, tandis que l’attribut du nième élément de la partie droite est nommé $n

Remarques :

* Un bloc d’actions peut apparaître au début et/ou au milieu de la partie droite de la règle
* Ces actions sont exécutées après la reconnaissance des symboles les précédant et avant la reconnaissance des symboles suivants
* Attention, un bloc d’action intermédiaire est comptabilisé comme un autre symbole dans la numérotation des attributs $i
* En effet, un bloc intermédiaire est lui-même associé à un attribut $n correspondant à sa position dans la partie droite
* A l’intérieur du bloc intermédiaire, la valeur de l’attribut associé à ce bloc peut être défini en affectant l’attribut $$
* Le type d’un bloc intermédiaire ne peut qu’être explicitement donné lors de son utilisation par : $<typeBloc>$ ou $<typeBloc>n
* Le typeBloc pouvant être n’importe lequel des types définis par YYSTYPE

**Exemple** :

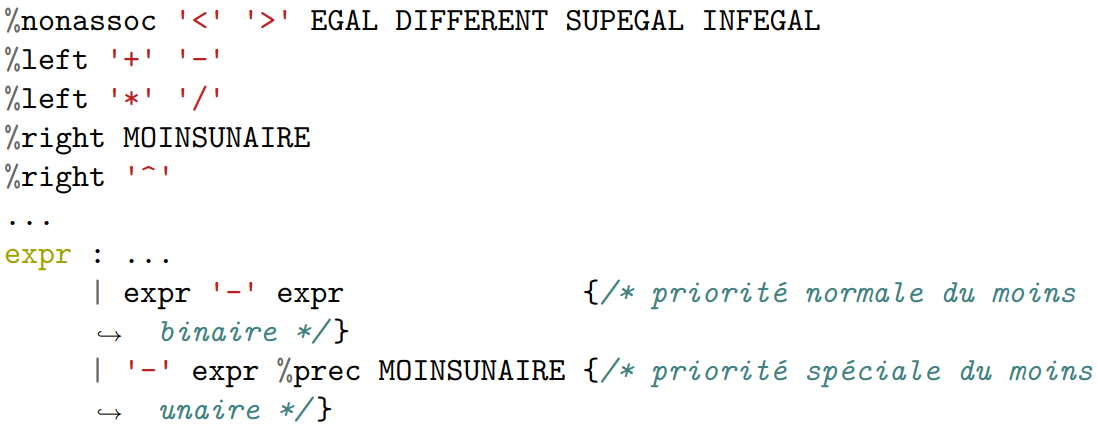
| bloc : '{' {initPourDeclarations();} decls insts '}'  | '{' insts '}'  ; |
| --- |

Dans cet exemple, le symbole non terminal decls a un attribut référencé par $3.

**Actions prédéfinies** :

| **$$** | attribut du non terminal en partie gauche de règle |
| --- | --- |
| **$n** | attribut associé au n ième composant de la partie droite |
| **$<typeAutre>n** | permet de spécifier un autre type que le type par défaut du n ième composant |
| **YYABORT** | retourne immédiatement de yyparse avec un résultat 1 (erreur) |
| **YYACCEPT** | retourne immédiatement de yyparse avec un résultat nul 0 |
| **YYBACKUP(jeton, valeurAttribut)** | dépile un jeton de l’automate |
| **yychar** | variable entière contenant le jeton de prévision courant |
| **YYEMPTY** | valeur stockée dans yychar quand il n’existe pas de jeton de prévision |
| **YYERROR** | provoque une erreur de syntaxe immédiate |
| **YYRECOVERING** | variable valant 1 si on est dans une récupération d’erreur, 0 sinon |
| **yyclearin** | supprime le le jeton de prévision courant |
| **yyerrok** | force le retour de la récupération d’erreur vers l’état normal de l’analyseur syntaxique. Appel après retour à la ligne. |

**Associativité et priorité des opérateurs** :

Dans la partie déclaration, on peut définir des priorités d’opérateurs et les règles définissant leur type d’associativité. Rappelons qu’un opérateur binaire infixe \* est associatif à gauche

(“left”) lorsque x ∗ y ∗ z = (x ∗ y) ∗ z et associatif à droite (“right”) lorsque x ∗ y ∗ z = x ∗ (y ∗ z). Lorsqu’un opérateur est associatif à gauche et à droite, il faudra choisir l’une des deux associativités pour indiquer l’ordre d’évaluation des expressions. Si un opérateur est non associatif, c’est-à-dire x ∗ y ∗ z n’est pas défini, il faudra également l’indiquer à bison par %nonassoc. On utilise de même %right et %nonassoc pour l’associativité à droite et la non associativité.

L’automate à pile choisit l’opération Shift ou Reduce en comparant la priorité de la règle courante avec celle du jeton de prévision. Si le jeton est plus prioritaire alors un Shift est effectué, sinon un Reduce est effectué. La priorité d’une règle est la priorité de son jeton le plus à droite.

**Interface avec lex** :

* yyparse() appelle itérativement yylex() jusqu’à ce que celui-ci retourne un jeton inférieur ou égal à 0.
* Les noms de jetons nommés peuvent être partagés par l’intermédiaire du fichier y.tab.h qui est automatiquement généré lorsqu’on utilise l’option -d de bison
* La valeur sémantique (attribut) d’un jeton sera passée de flex à bison par l’intermédiaire de la variable yylval qui est de type YYSTYPE

**Analyse ascendante par automate à pile** :

Nous allons étudier l’analyse ascendante et plus particulièrement

l’analyse LALR utilisée dans bison. Rappelons que, partant d’un mot (flot de jetons), on essaie de construire l’arbre de dérivation associé. Cette construction va se faire depuis les feuilles (jetons) en remontant jusqu’à la racine (l’axiome).

De plus, on va construire une dérivation droite (Rightmost) et à l’envers !

Les grammaires pouvant être analysées par un analyseur LR doivent, bien entendu, avoir certaines propriétés comme la non ambiguïté

| **Définition** : Un manche d’un mot (pas forcément terminal) m = αβγ est un couple constitué  d’une production X → β, d’une position p dans m telle que m[p, p + |β|[= β ;  ayant la propriété suivante : S∗⇒d αXγ 1⇒d m = αβγ. |
| --- |

Exemple : Soit la grammaire G = ({1, 2, 3, +}, {E}, R, E) avec les règles de R

suivantes : E → 1|2|3|E + E :

* Considérons le mot d’entrée 1+2+3$
* L’analyse du mot commence sur le 1 (Left to right scanning)
* Après avoir empilé (Shift) ce symbole, la règle E → 1 est appliquée et on empile E
* Arrivé sur le +, l’analyseur empile ce symbole car il ne peut pas appliquer de règle
* Le 2 est ensuite reconnu comme partie droite de E → 2
* On empile donc E et on s’aperçoit qu’on peut alors réduire (Reduce) le mot sur la pile (E+E) en appliquant la règle E → E + E
* La pile ne contient donc plus que E
* En continuant le même procédé, on reconnaît les productions E → 3 puis E → E + E
* On a donc la dérivation droite, obtenue à l’envers :

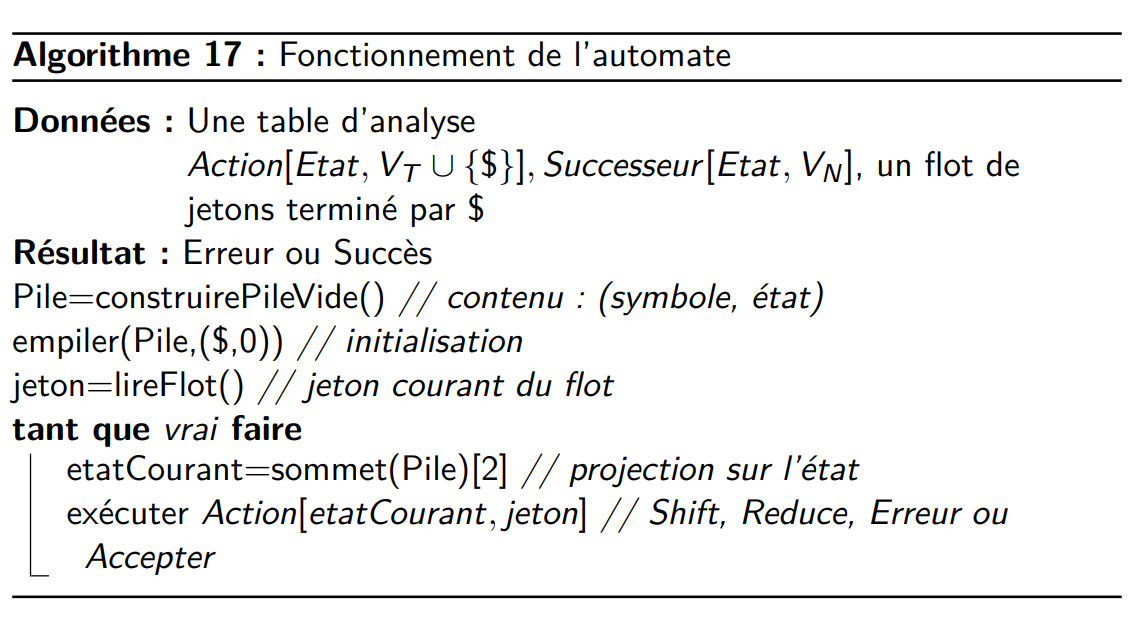
E(1)⇒E→E+E E + E ⇒E→3 E + 3 (1)⇒E→E+E E + E + 3 (1)⇒E→2 E + 2 + 3 (1)⇒E→1 1 + 2 + 3

| **Définition** : La pile d’un analyseur LR est une structure Dernier Entré Premier Sorti (LIFO) de couples (s,e) où s ∈ V ∪ {$} est un symbole et e ∈ N est un état entier. L’état courant de l’analyseur est l’état situé au sommet de la pile. |
| --- |

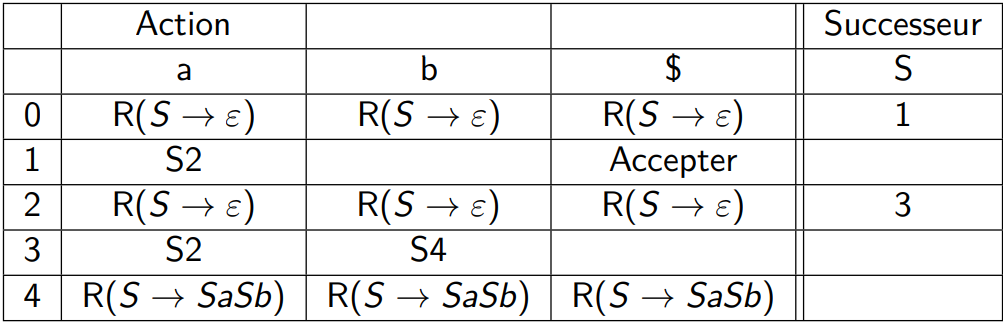
**Analyse LR :**

La table d’analyse d’un analyseur LR est constituée d’une partie Action et d’une partie Successeur. La table d’action est un tableau à deux entrées : les différents états sur les lignes, les terminaux et $ sur les colonnes. On note une case de cette table par Action[e, x]. Une action d’un analyseur LR peut être :

* **Décaler** (Shift) le symbole courant du flot d’entrée sur la pile (empiler) avec un état e. Cette action est notée : Se.
* **Réduire** (Reduce) par une production X → α. Cela consiste à dépiler α (à l’envers) de la pile et à le remplacer par X et l’état correspondant dans la table Successeur, c’est à dire Successeur[sommet(Pile)[2], X]. Cette action est notée : R(X → α).
* **Accepter** le mot d’entrée et terminer l’analyse. Cette action est notée : Accepter.
* **Générer** un message d’erreur de syntaxe et terminer l’analyse. Cette action n’est pas notée explicitement : toutes les cases vides de la table Action représentent des actions Erreur.

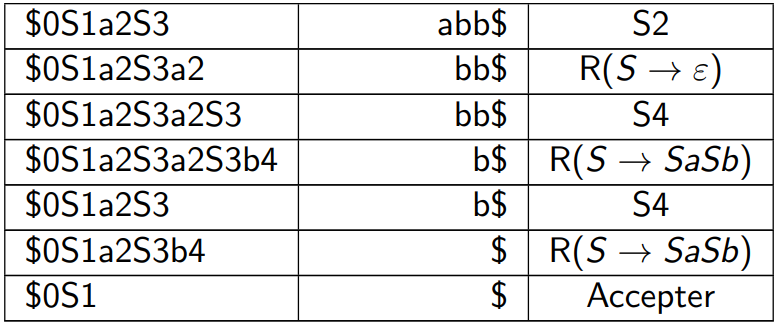
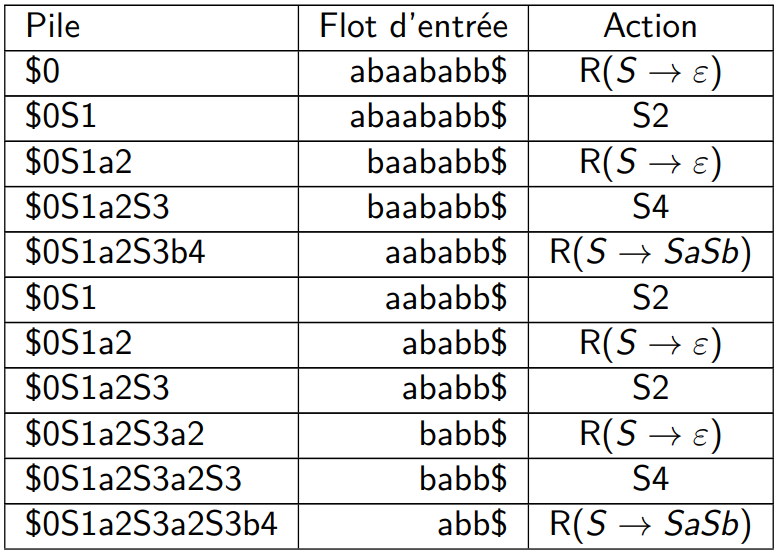
**Table d’analyse d’un analyseur LR :** 

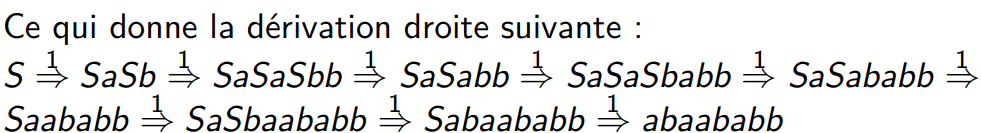
La table des successeurs est un tableau à deux entrées : les différents états sur les lignes, les non terminaux sur les colonnes. On note une case de cette table par Successeur[e, X]. Cette table ne sert qu’à indiquer le nouvel état courant après une réduction. Là aussi, toutes les cases vides de la table Successeur représentent des erreurs.

**Exemple** : Pour illustrer le fonctionnement de l’algorithme 17, prenons un exemple simple d’une grammaire de Dyck à un couple de parenthèses : Soit la grammaire Gd = ({a, b}, {S}, R, S) avec les règles de R suivantes : **S → SaSb|ε**

Le calcul des tables de cette grammaire fournit le résultat suivant :

Examinons l’analyse du mot abaababb$ :



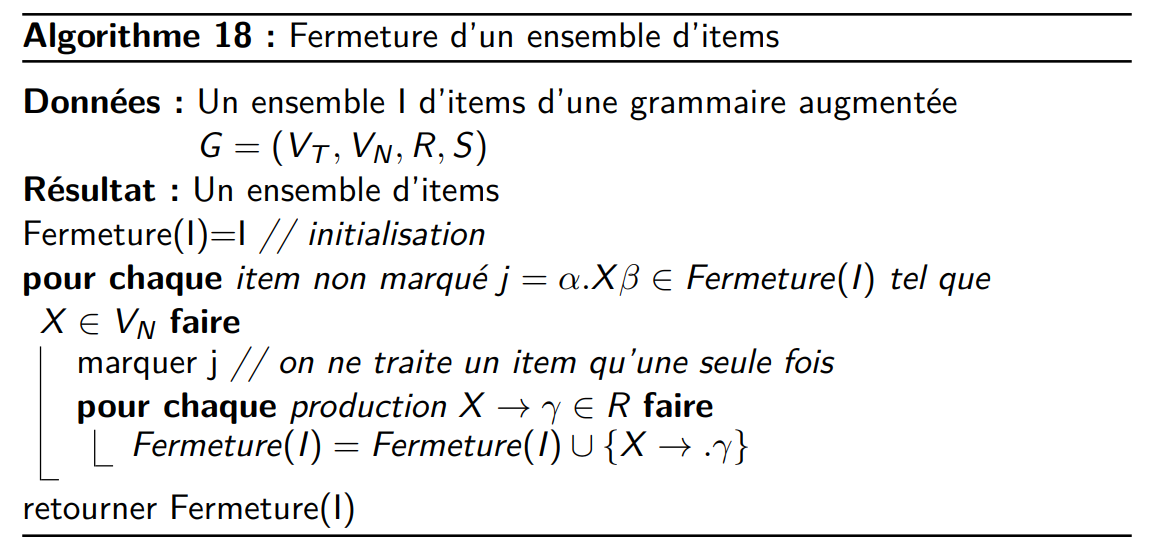


Nous allons décrire comment calculer les tables d’analyses pour des grammaires LR(1), c’est-à-dire avec un symbole de prévision Il existe plusieurs méthodes de construction dépendant de la complexité de la grammaire et de l’efficacité de l’analyseur, notamment en ce qui concerne la taille des tables.

| **Définition** : Un item LR(0), ou SLR, ou plus simplement item, d’une grammaire G = (VT , VN, R, S) est un couple constitué d’une production de R et d’une position dans la partie droite de celle-ci. La position est représentée par un point ’.’ dans la partie droite. |
| --- |

Soit la grammaire Gd = ({a, b}, {S}, R = {S → SaSb|ε}, S) L’ensemble des items de G est Items(G) = {S → .SaSb, S → S.aSb, S → Sa.Sb, S → SaS.b, S → SaSb., S → ε.} Un item représente ce qui a déjà été reconnu (à gauche du point) lors de l’analyse, et ce qu’il reste à reconnaître (à droite du point) avant de pouvoir réduire.

| **Définition** : Une grammaire augmentée G’ d’une grammaire G = (VT, VN, R, S) est  obtenue par ajout d’un nouvel axiome S’ et d’une production S’ → S :  G’ = (VT , VN ∪ {S’}, R ∪ {S’ → S}, S’) |
| --- |

**Construction de la collection canonique SLR** :

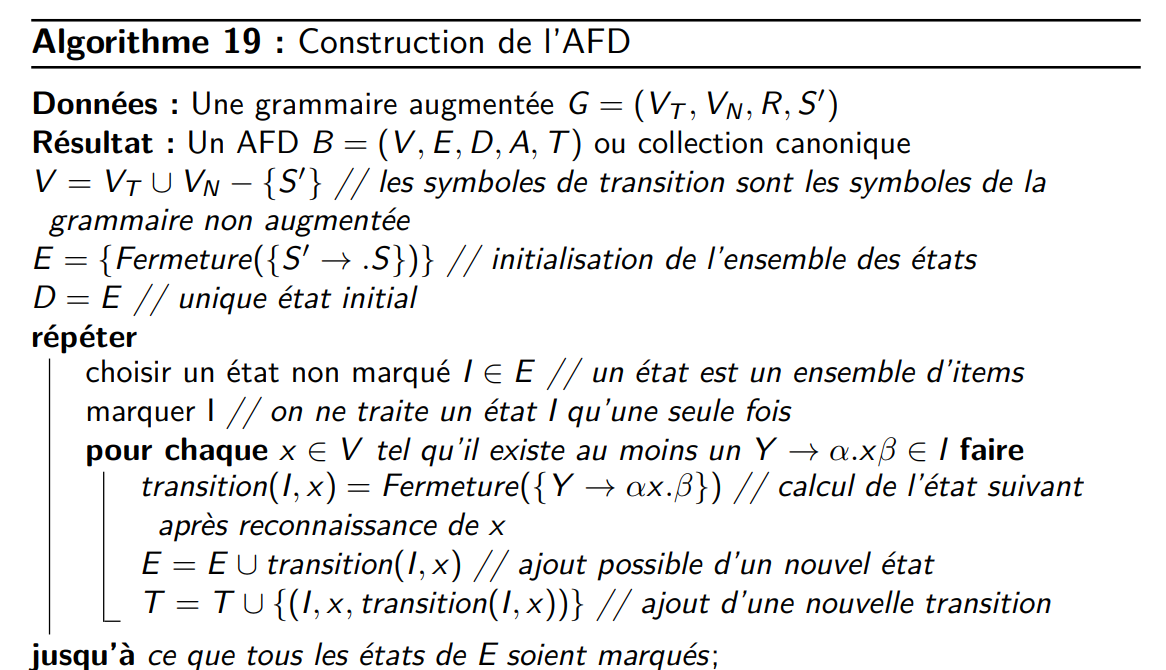
La construction de l’AFD utilise une fonction Fermeture() qui regroupe tous les items auxquels on peut s’attendre dans un état donné. Le principe de l’algorithme 18 tient en ce que lorsqu’on s’attend à reconnaître un non terminal X, il faut également s’attendre à reconnaître toute partie droite de production dont X est la partie gauche.

**Exemple** :

Soit la grammaire de Dyck augmentée : G’ = ({a, b}, {S, S’}, {S → SaSb|ε, S’ → S}, S’). Calculons les fermetures des ensembles d’items {S’ → .S} et {S → Sa.Sb}.

Fermeture({S’ → .S}) = {S’ → .S, S → .SaSb, S → ε.}

Fermeture({S → Sa.Sb}) = {S → Sa.Sb, S → .SaSb, S → ε.}



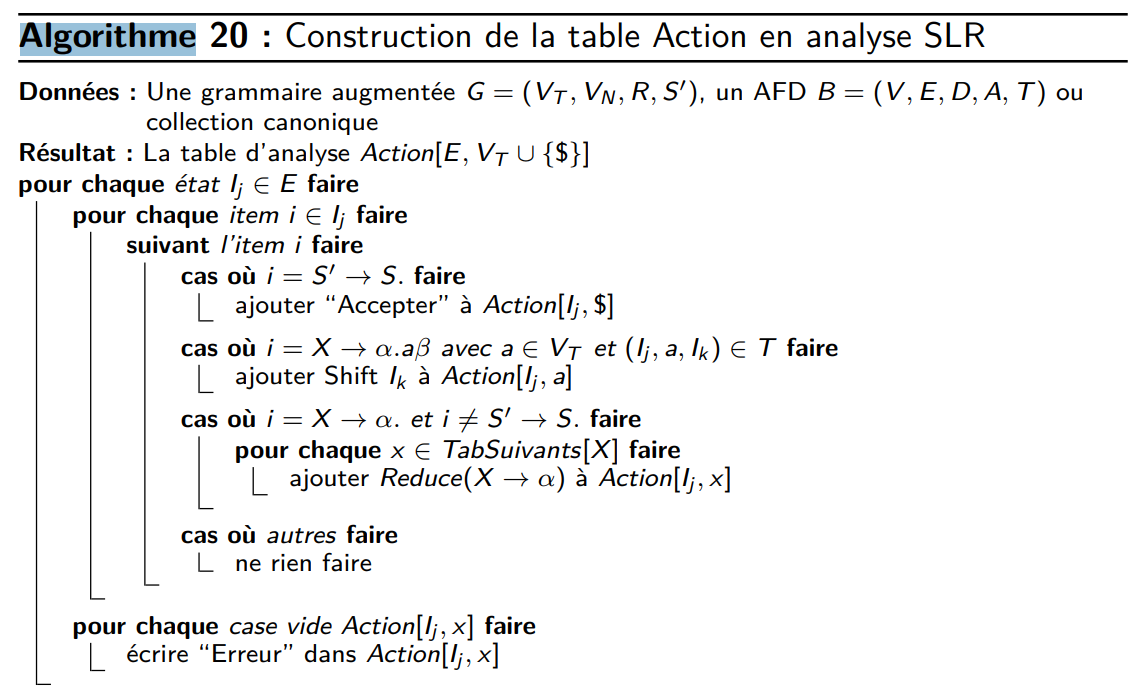
Pour construire l’AFD des états de l’analyseur, également appelée collection canonique des ensembles d’items LR(0), il faut examiner toutes les transitions possibles d’un état (ensemble d’items) vers un autre par le déplacement du “.” d’une position vers la droite.

**Remarques** :

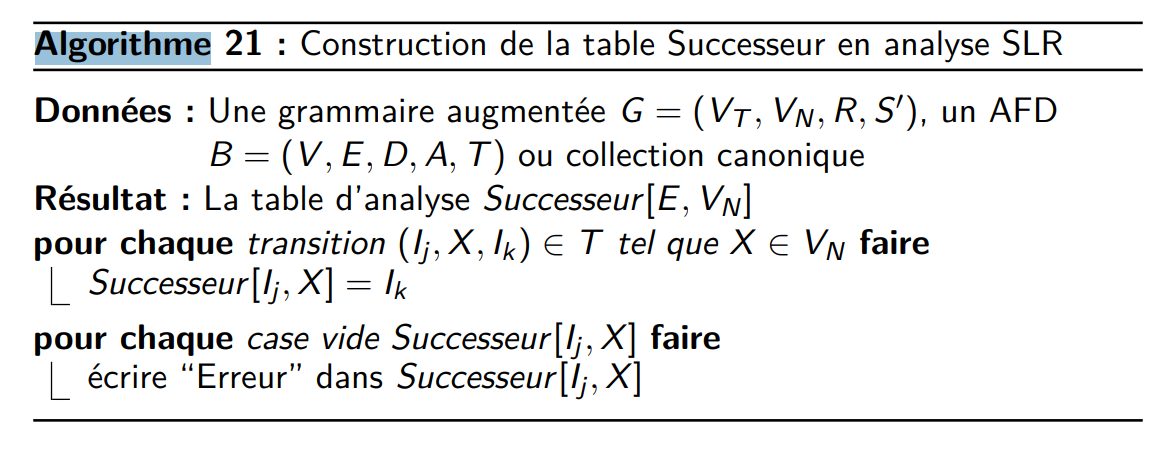
* Remarquons que l’algorithme 19 ne calcule pas d’états d’arrivée de l’automate
* En effet, cet automate ne permet pas de reconnaître un mot du langage analysé mais sert uniquement à décrire les transitions entre états
* Chaque chemin dans l’AFD correspond à un préfixe d’un mot dérivant de l’axiome
* Ces préfixes, aussi appelé préfixes viables, sont constitués de terminaux et de non terminaux
* Ils représentent le contenu possible de la pile de l’automate à un instant donné

| **Définition** : Un item X → β1.β2 est valide pour un préfixe αβ1 d’un mot dérivant de  l’axiome si et seulement s’il existe une dérivation droite : S’ ∗⇒d αXm 1⇒d αβ1β2m avec m ∈ VT∗, X ∈ VN, αβ1β2 ∈ V∗. |
| --- |

| **Théorème** : L’ensemble des items valides pour le préfixe viable αβ1 est l’ensemble des  items atteint par un parcours de l’AFD depuis l’état initial, le long du chemin étiqueté par αβ1. |
| --- |



**Exemple** : Soit le préfixe viable SaS, les deux items valides sont S → SaS.b et S → S.aSb. On a donc les deux types de dérivations droites possibles : S (1)⇒ SaSb ou bien S (1)⇒ SaSb (1)⇒ SaSaSb ∗⇒ SaSa . . . Remarquons que le symbole d’entrée suivant (a ou b) permettra de choisir l’état suivant qui correspondra soit à une réduction par S → SaSb ou bien par S → ε.



Remarquons qu’une seule action Accepter existe qui correspond à la réduction S’ → S de la grammaire augmentée. Une case de la table Action peut contenir plusieurs actions ! On peut obtenir des conflits Shift/Reduce ou Reduce/Reduce Dans ce cas, la grammaire n’est pas SLR et il sera nécessaire d’utiliser un algorithme de construction de table plus complexe.

Remarquons qu’il ne peut y avoir de conflit car l’automate est déterministe La table Successeur permet de déterminer l’état courant après une réduction en fonction de l’état sous-jacent dans la pile.

| **Théorème** : Une grammaire est LR(0) ou SLR si et seulement si sa table Action ne  contient aucun conflit. |
| --- |

| **Théorème** : Un langage est LR(0) ou SLR si et seulement s’il existe une grammaire  SLR le générant |
| --- |

* L’action Shift a bien été privilégiée par rapport aux deux reduce possibles
* bison parvient donc à fournir un analyseur pour nombre de grammaires mais attention, cet analyseur ne reconnaît que le mot aac, ce qui n’est pas correct vis à vis de la grammaire (ni aab, ni aaa ne sont reconnus)
* Pour finir, remarquons que certaines grammaires LR(1), c’est-à-dire nécessitant un seul jeton de prévision, ne sont pas analysables avec la méthode LALR.
* Bison avec l’option verbose -v fournit un fichier texte d’extension output vu auparavant
* Avec l’option graph -g, il fournit également un graphe graphviz d’extension gv illustrant la collection canonique, les conflits et leur résolution !
* Le graphe du transparent suivant a été obtenu par la commande :

$ bison -yvg calcAvecConflits.y

* La grammaire calcAvecConflits.y contient :

expr : '(' expr ')' {$$ = $2;}

| expr '+' expr {$$ = $1 + $3;}

| expr '\*' expr {$$ = $1 \* $3;}

| LITFLOT {$$ = $1;}

;

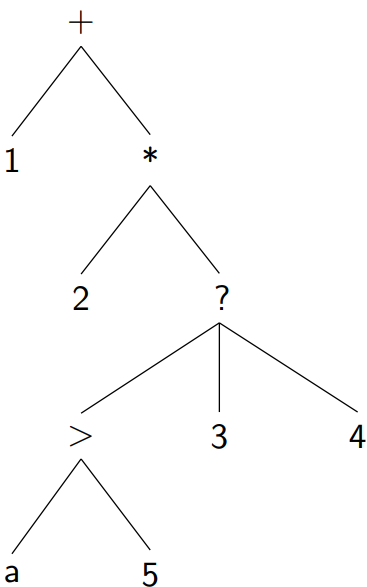
* Elle génère 4 conflits S/R !

**AST ou arbre syntaxique abstrait :**

Un arbre dont les nœuds internes sont marqués par :

* des opérateurs tels + ou \*
* ou des noms de structures de contrôle ou de structures syntaxiques tels while ou block

et dont les feuilles (ou nœuds externes) représentent :

* des opérandes tels un identificateur ou un littéral
* ou des instructions élémentaires telle ’ ;’ (instruction vide)

L’AST de gauche représente l’expression entière : 1 + 2 ∗ (a > 5?3 : 4)

Remarquons que l’AST est une simplification de l’arbre de dérivation (plus de parenthèses ni de symboles non terminaux)

l’ajout d’attributs à des noeuds de l’AST est permis durant tout le

processus d’analyse (e.g. type de l’expression inféré pendant l’analyse

sémantique) l’AST ainsi attribué est souvent appelé arbre décoré !

* Chaque bloc d’instruction doit donc être associé à une table des symboles (identificateurs) permettant d’accumuler l’information sur les symboles locaux à ce bloc (nom, type, valeur initiale, adresse ...)
* lors de reconnaissance syntaxique d’un identificateur, la liaison (binding) entre ce dernier et l’entrée dans une table des symboles le représentant peut être complexe voire impossible suivant le langage
* dans la plupart des langages interprétés, la liaison sera dynamique car le symbole sera recherché en remontant les contextes d’exécution et leurs tables des symboles :

Arrêt diapo 304

<https://complex-systems-ai.com/theorie-des-langages/construction-de-thompson/>

<https://pages.lip6.fr/Jean-Francois.Perrot/inalco/Automates/AnSynt/DescRec/DescRec.html#:~:text=mardi%2028%20f%C3%A9vrier.-,Principe%20de%20l'automatisation%20de%20la%20descente%20r%C3%A9cursive,l'axiome%20de%20la%20grammaire>.

<https://pages.lip6.fr/Jean-Francois.Perrot/inalco/Automates/Cours20.html>