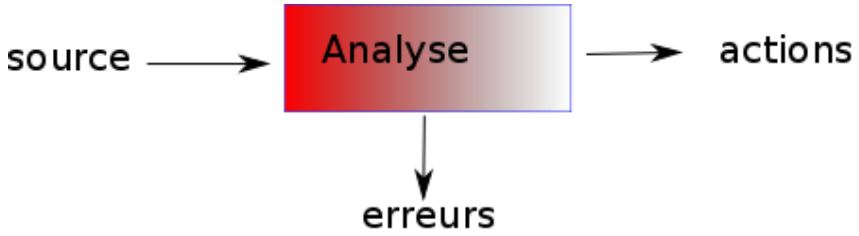
Compilation

Luc Brun

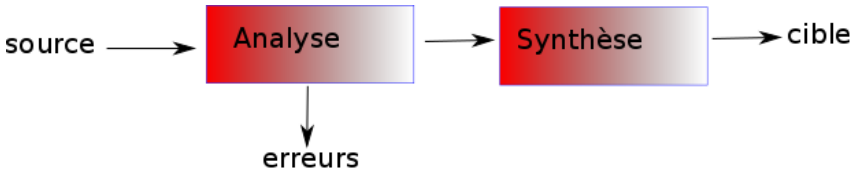
Plan

Définition

- La compilation est l'analyse automatique d'un langage (avec un vocabulaire et une syntaxe).



- Action :
  - Interpréteurs (javascripts, python, shell,SQL,. . .),
  - editeurs structurels,
  - controleurs statiques.
- Synthèse

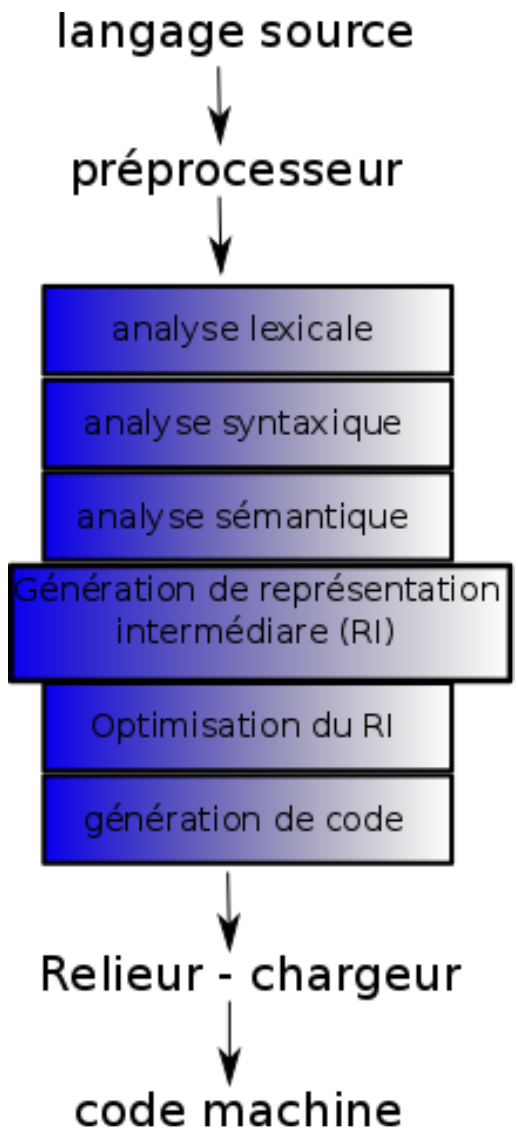


  - Compilateurs
  - Filtres

La compilation va au delà du domaine (déjà très large) de la génération de code exécutables à partir de langages sources. Elle pose la question des langages compréhensibles sans ambiguités.

Structure d'un compilateur

- Analyse lexicale : vérifie/reconnait le vocabulaire (identificateurs, mots clés,. . .)
- Analyse syntaxique : vérifie/reconnait la grammaire
- Analyse sémantique : vérifie le sens du programme (e.x. les types)
- RI : Langage intermédiare permettant de factoriser des étapes pour plusieurs langages.



Bibliographie

- COMPILATEURS, Principes, techniques et outils. Alfred Aho, Ravi Sethi, Jeffrey Ullman. InterEditions.
- Les compilateurs, théorie, construction, génération. R. Wilhelm, D. Maurer. Masson Eds.
- http ://uuu.enseirb-matmeca.fr/∼dbarthou/compilation/courscompilation.pdf

Analyse lexicale

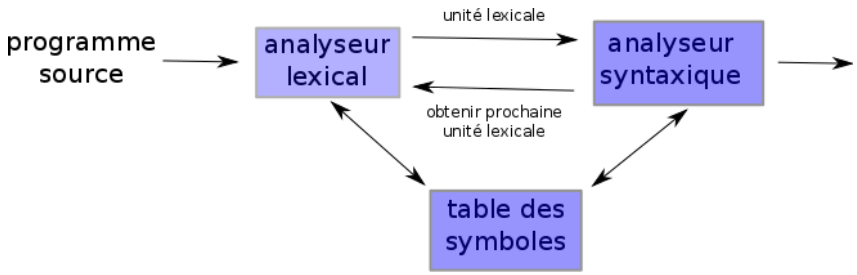
Objectifs :

- Transformation d'un ensemble de caractères en concepts (nombres, identificateurs, mots clés,...)
- On distingue les concepts suivants :

Unité lexicale : correspond à une entité (un concept) renvoyé par l'analyseur lexical. Par exemple $<, >, <=, >=$ sont tous des opérateurs relationels.

Un lexème est une instance d'unité lexicale. Par exemple le lexème 6.28 une instance de l'unité lexicale nombre.

Un modèle associe des lexèmes à leur unité lexicale.



Alphabet :

- Un alphabet Σ est un ensemble fini de symboles. L'alphabet binaire $\{0, 1\}$ et les codes ASCII sont des exemples d'alphabets, $\{A, G, C, T\}$ est l'alphabet des bases ADN.
- Une chaîne, est une séquence finie de symboles. Le terme mot est également utilisé.
- Un préfixe de s est obtenu en supprimant un nombre quelconque de caractères en fin de s.
- Un suffixe de s est obtenu en supprimant un nombre quelconque de caractères en début de s.
- Une sous-chaîne de s est obtenue en supprimant un préfixe et/ou un suffixe de s
- Un préfixe, suffixe, sous chaîne *propre de s est un suffixe, préfixe ou une sous chaîne de s non égal à s.*
- *Sous suite de s : toute chaîne obtenue en supprimant des caractères de s.*

Langages :

- Un langage est un ensemble a priori quelconques de chaînes construites sur un alphabet. L'ensemble des codes source C, des nombres binaires ou des codes ADN sont des langages.
- $\emptyset$ désigne le langage vide. On désigne également par $\{\epsilon\}$ le langage composé uniquement de la chaîne vide.
- Si x et y sont deux chaînes, la concaténation de x et y, xy représente la chaîne formée par la séquence des symboles de x suivie de celle de y. Ex : x=anti, y=constitutionnellement, xy=anticonstitutionnellement.
- Un alphabet Σ munit de la concaténation et de son élément neutre ϵ a une structure de monoïde (intuitivement un groupe sans l'inverse).
- Exponentiation : $s^0 = \epsilon$, $s^1 = s$, $s^n = s^{n-1}s$

Opérations sur les langages

Définitions :

- Union $(L \cup M)$:

$$L \cup M = \{s \mid s \in L \text{ ou } s \in M\}$$

- Concaténation (LM) :

$$LM = \{st \mid s \in L \text{ et } s \in M\}$$

Remarque : LL est dénoté L^2, $L^0 = \{\epsilon\}, L^1 = L$.

- Fermeture de Kleene (L^*) :

$$L^* = \bigcup_{i=0}^{+\infty} L^i$$

- Fermeture positive (L^+) :

$$L^+ = \bigcup_{i=1}^{+\infty} L^i$$

Opérations sur les langages

Exemples :

- Si $B = \{0, 1\}$,
  B^+ est l'ensemble des nombres binaires d'au moins un chiffre.
- Si $C = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
  C^4 est l'ensemle des nombres de 4 chiffres.
- Si $K = \{a, \ldots, z, A, \ldots, Z\}$,
  $K(K \cup C)^*$ est l'ensemble des mots commençant par une lettre puis composés d'un nombre quelconque de lettres et chiffres.

Expressions régulières

- Une expression régulière r est construite récursivement à partir d'union (notée $|$), de concaténation, et de fermeture. Elle définie un langage $L(r)$.

1. ϵ est une expression régulière qui dénote $L(\epsilon) = \{\epsilon\}$,
2. $a \in \Sigma$ est une expression régulière qui dénote l'ensemble $L(a) = \{a\}$,

2.1 (r) est une expression régulière dénotant $L(r)$,

2.2 $(r)|(s)$ est une expression régulière dénotant $L(r) \cup L(s)$,

2.3 $(r)(s)$ est une expression régulière dénotant $L(r)L(s)$,

2.4 $(r)^*$ est une expression régulière dénotant $L(r)^*$,

- Toute expression construite a partir de la construction ci-dessus est régulière.

Exemple

Soit $\Sigma = \{a, b\}$

- $L(a|b) = \{a, b\}$,
- $L((a|b)(a|b)) = \{aa, ab, ba, bb\}$,
- $L(a^*) = \{\epsilon, a, a^2, \dots\}$,
- $\{a, a^2, b, b^2, ab, ab^2, a^2b^2, \dots\} \subset L((a|b)^*)$,
- On évite des paranthèse inutiles en utilisant les conventions suivantes :
- $*$ a la plus haute priorité,
- la concaténation à la deuxième plus haute priorité et est associative à gauche,
- $|$ a la plus faible priorité et est associatif à gauche.

$$a^*(a|b|c)^*de = ((a)^*\,(((a)|(b))|(c))^*)((d)(e))$$

Propriétés algébriques

Axiome	Descriptif
$r\|s = s\|r$	$\|$ est commutatif
$r\|(s\|t) = (r\|s)\|t$	$\|$ est associatif
$(rs)t = r(st)$	la concaténation est associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	la concaténation est distributive vis à vis de $\|$
$\epsilon r = r\epsilon = r$	ϵ est un élémement neutre pour la concaténation
$r^* = (r\|\epsilon)^*$	
$r^{**} = r^*$	$*$ est idempotent

Définitions régulières

Il peut être commode de donner des noms à des expressions régulières et de s'en reservir dans des expressions plus complexes. On utilise pour cela des définitions régulières.

- Soit Σ un alphabet, et un ensemble de couples

$$\begin{array}{lcl} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \vdots & \\ d_n & \rightarrow & r_n \end{array}$$

où r_i est une expression régulière et d_i son nom associé.

- Cette définition est appellée régulière si chaque r_i représente une expression régulière construite sur $\Sigma \cup \{d_1, \ldots, d_{i-1}\}$.

Exemple de définition régulière

- Identificateurs de variables

$$\begin{array}{rcl} \textbf{lettre} & \rightarrow & a|b|c\ldots|z|A|B\ldots|Z \\ \textbf{chiffre} & \rightarrow & 0|1|2|3|4|5|6|7|8|9 \\ \textbf{id} & \rightarrow & \textbf{lettre}(\textbf{lettre}|\textbf{chiffre})^* \end{array}$$

- Nombres :

$$\begin{array}{rcl} \textbf{pm} & \rightarrow & (+|-) \\ \textbf{chiffre} & \rightarrow & 0|1|2|3|4|5|6|7|8|9 \\ \textbf{frac} & \rightarrow & .\textbf{chiffre}^+ \\ \textbf{exp} & \rightarrow & E\ (\textbf{pm}|\epsilon)\ \textbf{chiffre}^+ \\ \textbf{nb} & \rightarrow & (\textbf{pm}|\epsilon)\ \left(\textbf{chiffre}^+(\textbf{frac}|\epsilon)|\textbf{frac}\right)\ (\textbf{exp}|\epsilon) \end{array}$$

NB : 14. ou -E5 ne sont pas reconnus par cette définition.

Notations abrégées et limites des expressions régulières

Notations abrégées :

- L'expression $(r|\epsilon)$ se note également $r?$. Ainsi **exp** $\rightarrow E$ $(\textbf{pm}|\epsilon)$ $\textbf{chiffre}^+$, se note également **exp** $\rightarrow E$ **pm**? $\textbf{chiffre}^+$,
- $(a|b|c)$ se note également $[abc]$. De même $(a|b|\ldots,|z)$ se note $[a-z]$ et $[a-zA-Z0-9]$ représente $(a|\ldots,|z|A|\ldots|Z|0\ldots|9)$. Ainsi l'ensemble des identificateurs se note : $[a-zA-Z][a-zA-Z0-9]^*$.

Limites des expressions régulières :
Ayez bien conscience que les expressions régulières ne permettent de coder qu'un nombre limité de languages. En particulier, les expressions régulières ne permettent pas de compter. Ainsi le langage :

$$L = \{wcw \mid w \in L(\{a,b\}^*)\}$$

n'est pas régulier.

Unités lexicales et diagrammes de transition

Un diagramme de transition est une représentation graphique du processus de reconnaissance d'une unité lexicale. Il représente les actions réalisées par l'analyseur lexical sur le tampon d'entré lorsqu'il est appelé par l'analyseur syntaxique.

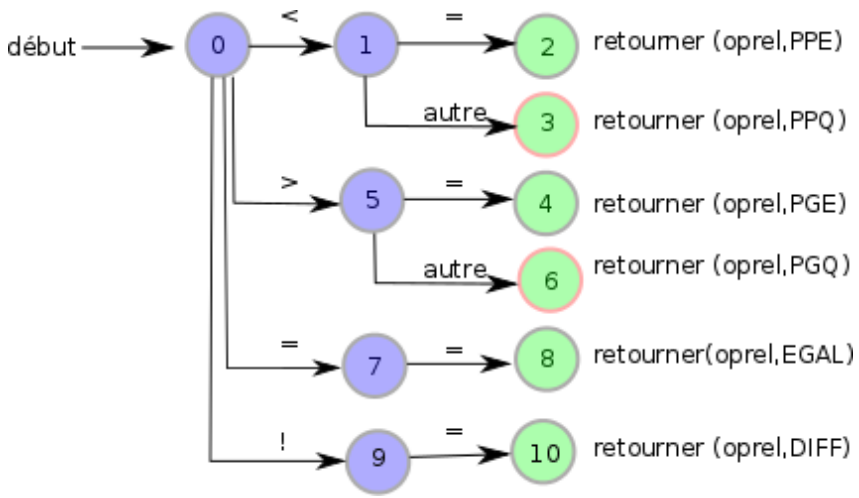
- Un diagramme de transition est constitué d'états :

- Internes 
- Finaux : 
- Finaux avec recul de tampon  Ces états d'acceptation codent les cas où il est nécessaire de reculer le tampon d'entré.

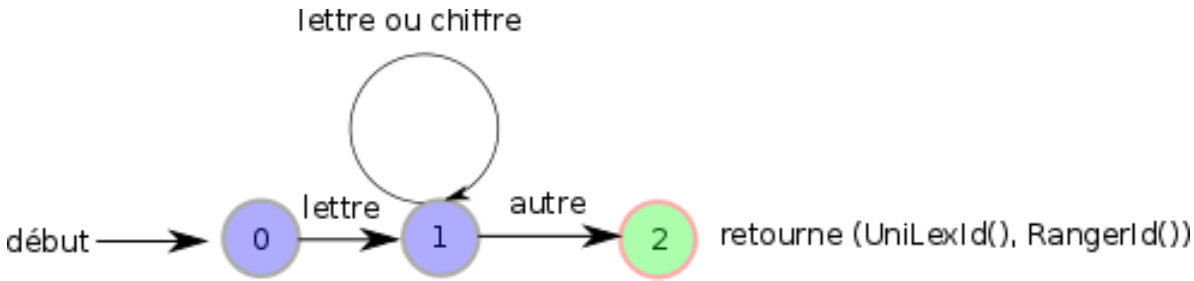
- Les arcs codent les transitions entre états. Ils ont des étiquettes indiquant sur qu'elle entrée s'effectue chaque changement d'état. L'étiquette autre, code tout caractère autre que ceux indiqués par les changements d'états sur l'état courrant. On suppose les diagrammes déterministes (i.e. une même étiquette ne peut envoyer sur deux états différents)

Diagramme de transition : Exemple

- **oprel** $\rightarrow (< \mid > \mid <= \mid >= \mid == \mid !=)$



- **id** $\rightarrow$ **lettre**(**lettre**|**chiffre**)*.



Automates fini non déterministes (AFN)

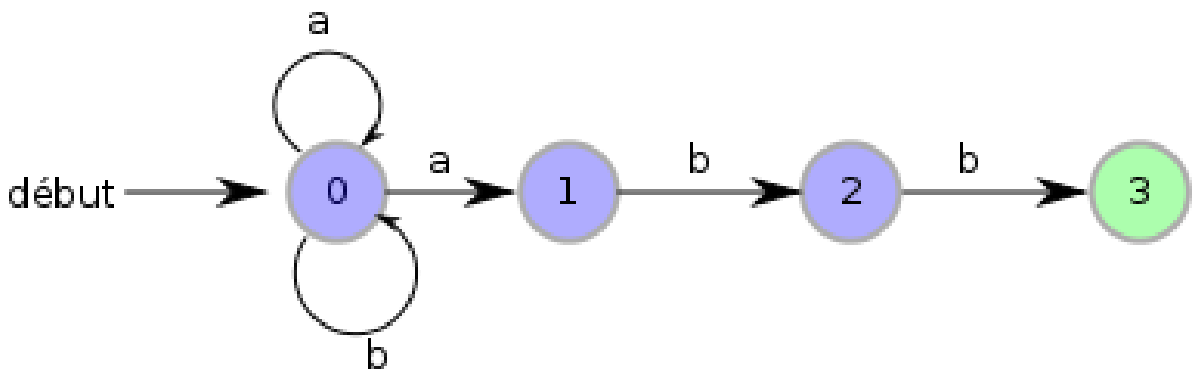
Un AFN est un modèle mathématique défini par :

- Un ensemble d'états E,
- Un ensemble de symboles d'entrées Σ
- Une fonction de transition Transiter, qui fait correspondre des couples (état,symbole) à des ensembles d'états
- Un état e_0 correspondant à l'état de départ
- Un ensemble d'états F représentant les états d'acceptation (ou états finaux).

Le langage défini par un AFN est l'ensemble des chaînes menant à un état d'acceptation.

AFN : Un premier exemple

- Soit l'unité lexicale $(a|b)^*abb$. Celle ci est reconnue par l'AFN suivant :

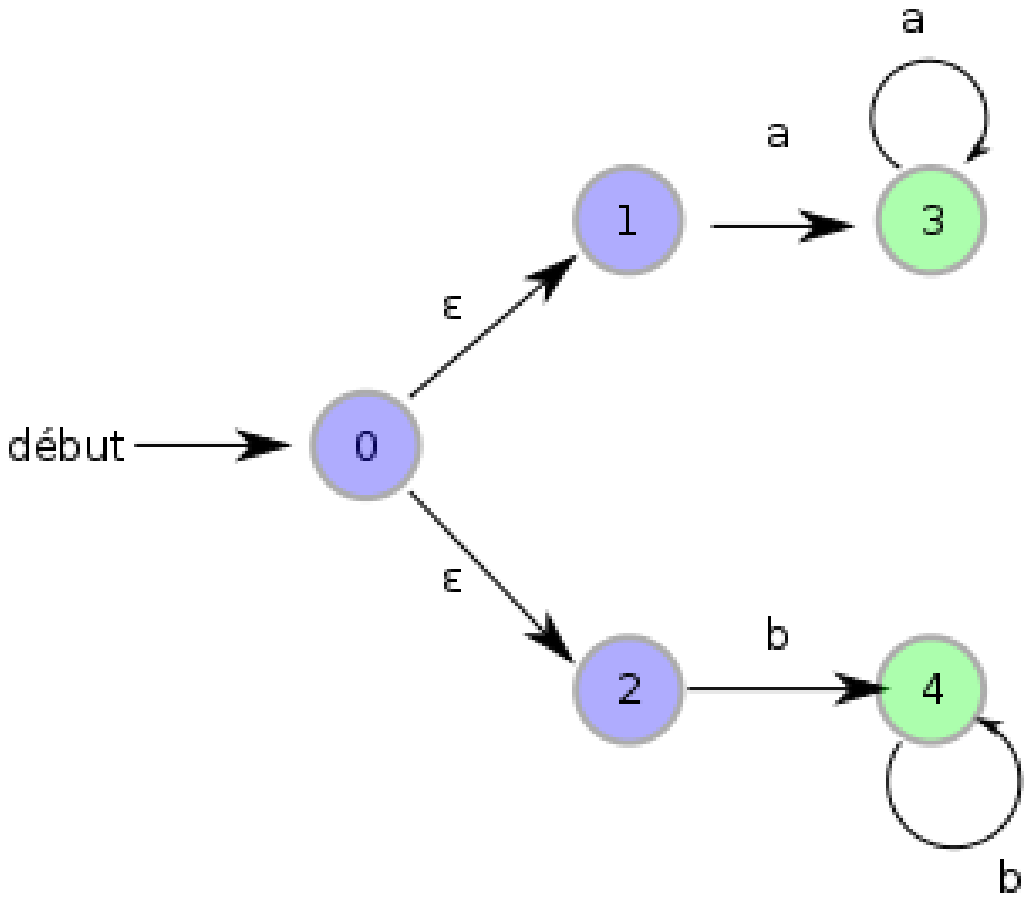


- associé à la table de transition :

Transiter		
Etat	Symbole d'entrée	
	a	b
0	$\{0, 1\}$	0
1	–	2
2	–	3

AFN : Un second exemple

- Soit l'unité lexicale $aa^*|bb^*$

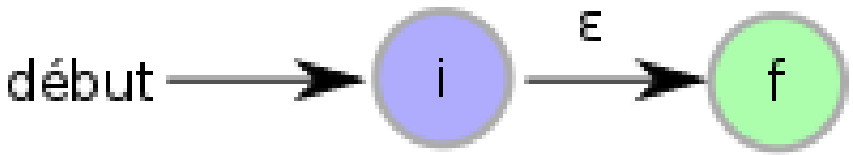


- Notez les transitions ϵ sur l'état de départ et les boucle sur les états d'acceptation.

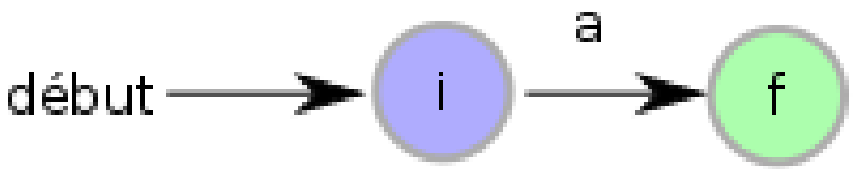
Construction d'un AFN à partir d'une expression régulière

Construction récursive :

1. Pour ϵ construire l'AFN :



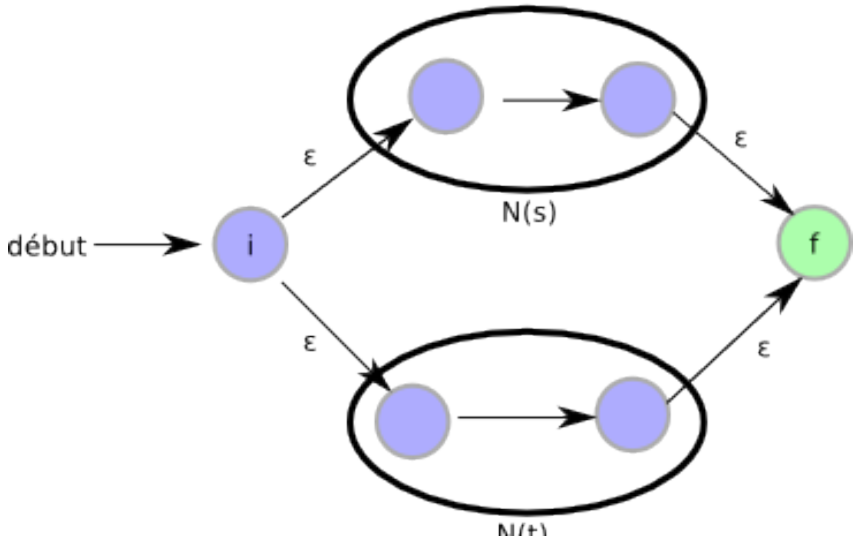
2. Pour $a \in \Sigma$ construire l'AFN :



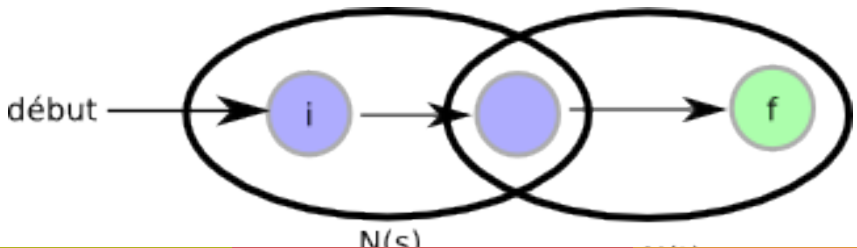
Construction d'un AFN à partir d'une expression régulière

Construction récursive :

3. Soient $N(s)$ et $N(t)$ les AFN des expressions s et t. L'AFN $N(s|t)$ est défini par :



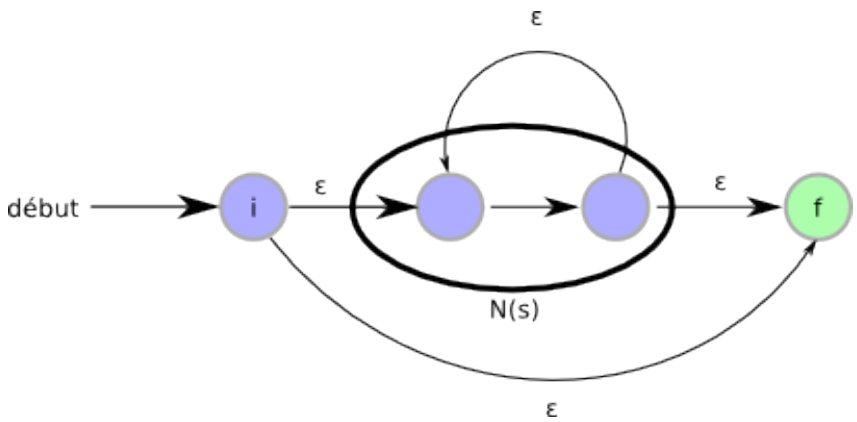
4. Soient $N(s)$ et $N(t)$ les AFN des expressions s et t. L'AFN $N(st)$ est défini par :



Construction d'un AFN à partir d'une expression régulière

Construction récursive :

4. Soit $N(s)$ l'AFN de l'expression s. L'AFN $N(s^*)$ est défini par :



5. L'AFN associé à (s) est $N(s)$ lui même.

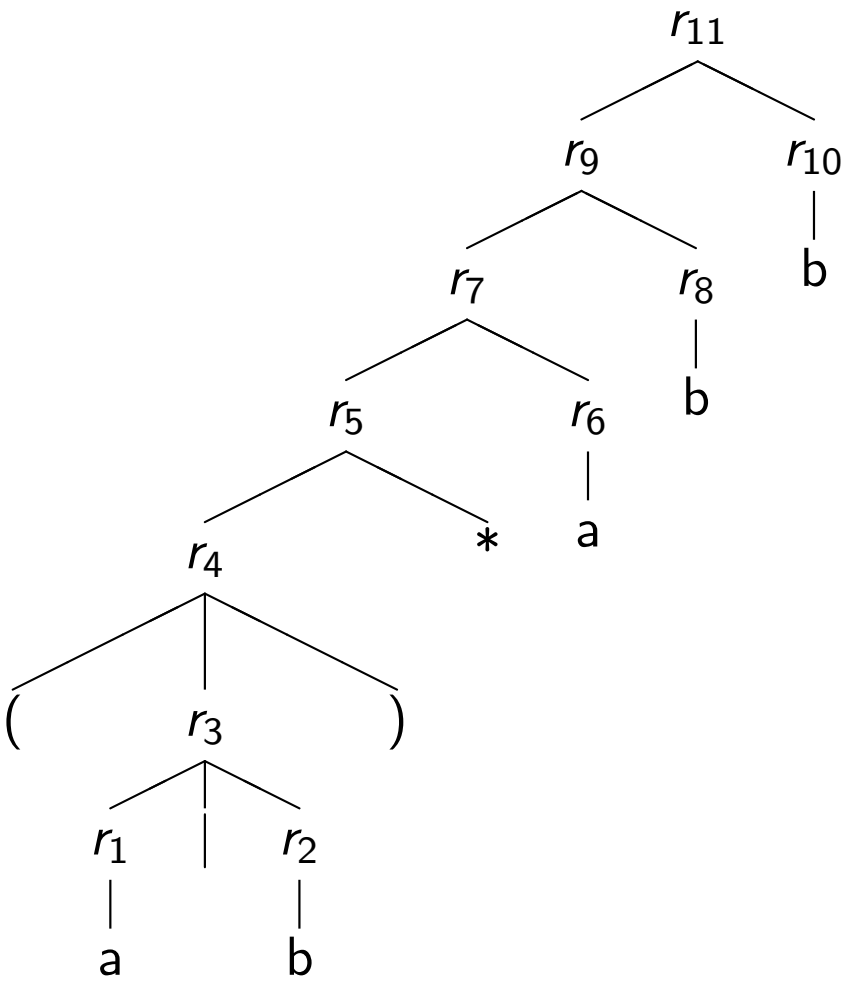
Propriétés de l'AFN construit

Les propriétés suivante se montrent facilement en suivant le processus de construction.

- $N(r)$ a au plus deux fois plus d'états qu'il n'y a de symboles dans r.
- $N(r)$ a exactement un état de départ et un état d'acceptation. L'état d'acceptation n'a pas de transition sortante.
- Chaque état de $N(r)$ a soit une transition sortante sur un symbole de Σ, soit au plus deux transistions sortantes sur ϵ.

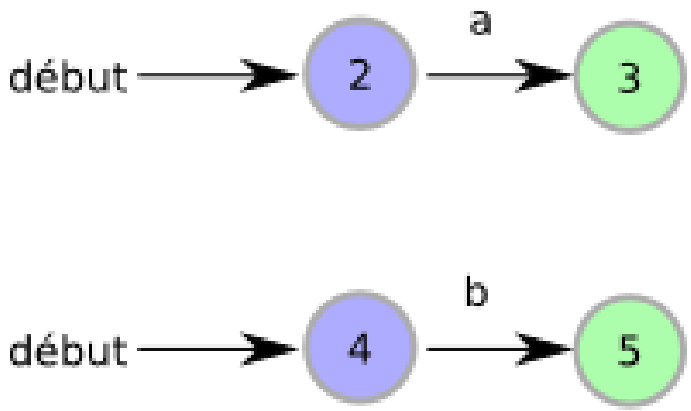
Exemple

- Évaluation de $(a|b)^*abb$. Son arbre syntaxique associé est (cf évaluation d'expressions cours 1A algo) :



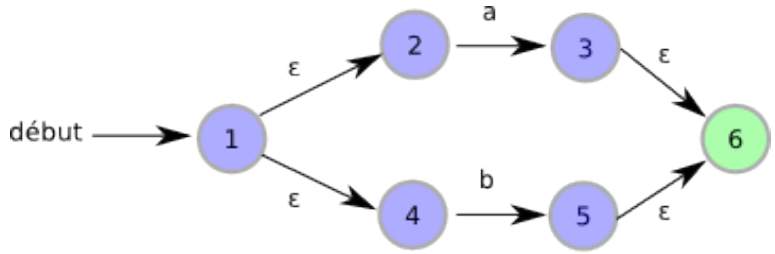
Exemple de construction d'AFN

- Construisons les AFN, $N(r_1)$ et $N(r_2)$:

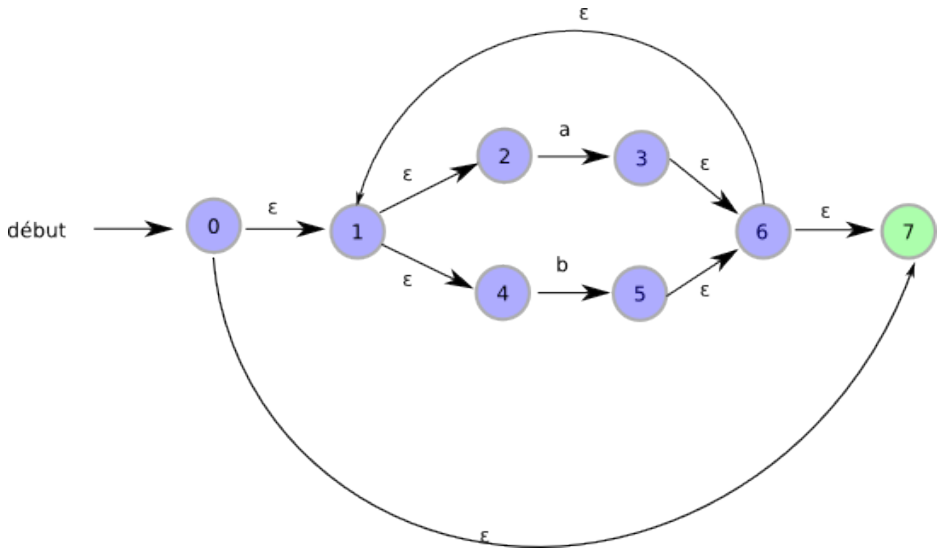


Exemple de construction d'AFN

- Construisons l'AFN de $r_3 = r_1|r_2$.

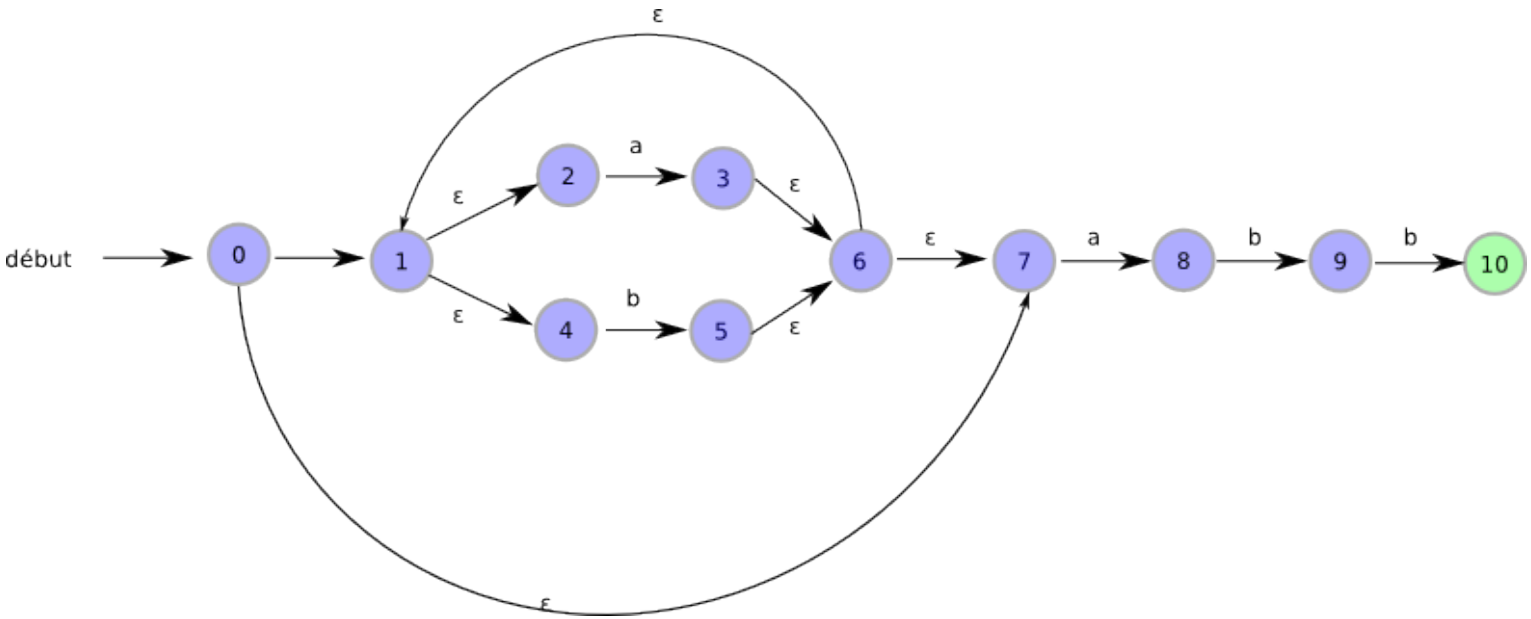


- $N(r_4) = N(r_3)$, construisons l'AFN de $r_5 = (r_1|r_2)^*$



Exemple de construction d'AFN

- Construisons directement r_{11} par 3 applications successives de la règle 4. Pour obtenir l'AFN final :



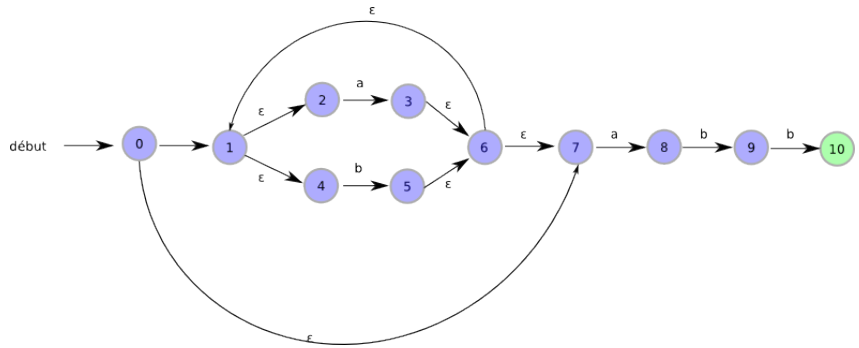
Reconnaissance d'une expression régulière par un AFN

L'algorithme calcule itérativement un ensemble d'états E en utilisant les fonctions :

- Transiter(E,a) : retourne l'ensemble des états accessibles via les états E par une transition 'a'.
- ϵ-fermeture(E) : renvoie l'ensemble des états accessibles depuis E par une ϵ transition.

fonction recAFN **(**e_0,F**) :** Booléen
    Déclaration E : Ensemble d'états,c : caractère
début
    E $\leftarrow$ ϵ-fermeture(e_0)
    c $\leftarrow$ carSuivant()
    tant que $c \neq$ fdf **faire**
        E $\leftarrow$ ϵ-fermeture(Transiter(E,c))
        c $\leftarrow$ carSuivant()
    fintantque
    retourner $E \cap F \neq \emptyset$
fin

Déroulement de la reconnaissance d'une expression régulière par un AFN



Considérons le lexème $s = aabb$.

1. E=ϵ-fermeture(0)=$\{0, 1, 2, 4, 7\}$; c='a'
2. Transiter(E,'a')=$\{3, 8\}$;
   ϵ-fermeture(Transiter(E,'a')=$\{1, 2, 3, 4, 6, 7, 8\}$;c='a'
3. Transiter(E,'a')=$\{3, 8\}$;
   ϵ-fermeture(Transiter(E,'a')=$\{1, 2, 3, 4, 6, 7, 8\}$;c='b'
4. Transiter(E,'b')=$\{5, 9\}$;
   ϵ-fermeture(Transiter(E,'a')=$\{1, 2, 4, 5, 6, 7, 9\}$;c='b'
5. Transiter(E,'b')=$\{5, 10\}$;
   ϵ-fermeture(Transiter(E,'a')=$\{1, 2, 4, 5, 6, 7, 10\}$;c='fdf'

Des AFN aux Automates finis déterministes (AFD)

L'utilisation d'un AFN pour reconnaître un lexème est parfaite. En revanche si on doit traiter des milliers ou des millions de chaînes il est certain que l'on effectuera de nombreuses fois les mêmes ϵ-fermeture et les mêmes transitions. Un automate fini déterministe (AFD) est un automate où :

- Aucun état n'a de ϵ-transition
- Pour chaque état e et chaque symbole $a \in \Sigma$ il existe au plus une transition étiquetée 'a' sortant de e.

La reconnaissance d'un lexème par un AFN consiste donc simplement a tester si il existe un chemin de l'état d'entré à un état d'acceptation.

Reconnaissance d'un lexème par un AFD

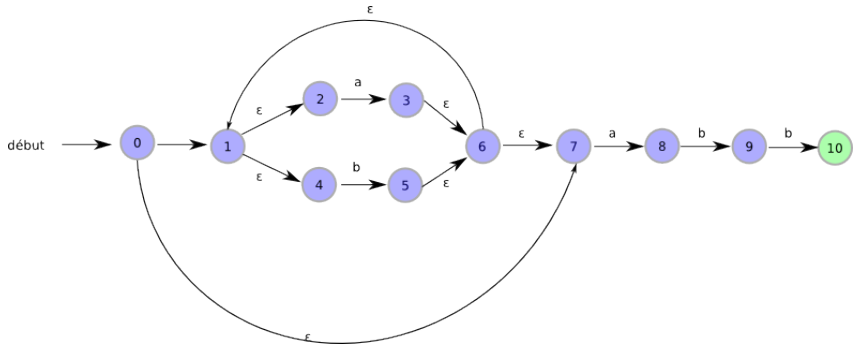
fonction recAFD **(**e_0,F**) :** Booléen
    Déclaration e : état,c : caractère
début
    e $\leftarrow$ e_0
    c $\leftarrow$ carSuivant()
    tant que $c \neq$ fdf **faire**
        e $\leftarrow$ Transiter(e,c)
        c $\leftarrow$ carSuivant()
    fintantque
    retourner $e \in F$
fin

Transformation d'un AFN en AFD

La reconnaissance d'un lexème par un AFN calcule une suite d'ensembles d'états. L'idée de base est de pré calculer ces ensembles d'états et de les stocker dans l'AFD avec leurs transitions (table Dtran).

fonction recAFD **(**AFN(e_0,F)**) :** AFD
    Déclaration Détat : ensemble d'états
début
    D-état $\leftarrow$ ϵ-fermeture($\{e_0\}$)
    tant que il existe T non marqué dans Détat **faire**
        marquer T
        Pour chaque a **dans** Σ **faire**
            U $\leftarrow$ ϵ-fermeture(Transiter(T,a))
            si $U \notin$ Détat **alors**
                Détat $\leftarrow$ Détat$\cup\{U\}$
            finsi
            Dtran[T,a] $\leftarrow$ U
        finpour
    fintantque
    retourner AFD(Dtran,Détat)
fin

Exemple de transformation

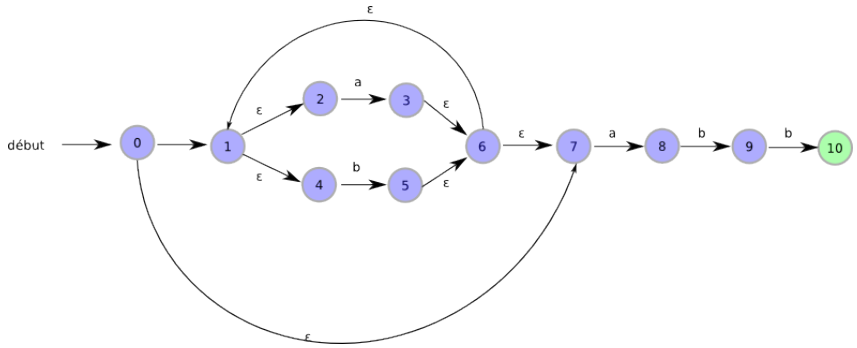


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A

Transiter		
Etat	Symbole d'entrée	
	a	b
A		

$$A = \{0, 1, 2, 4, 7\}$$

Exemple de transformation

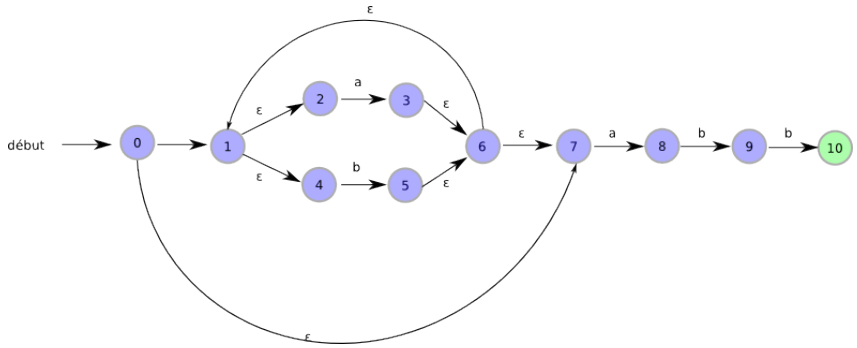


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(A,a))=ϵ-fermeture($\{3, 8\}$)=$\{1, 2, 3, 4, 6, 7, 8\}$=B

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	
B		

$$\begin{aligned} A &= \{0, 1, 2, 4, 7\} \\ B &= \{1, 2, 3, 4, 6, 7, 8\} \end{aligned}$$

Exemple de transformation

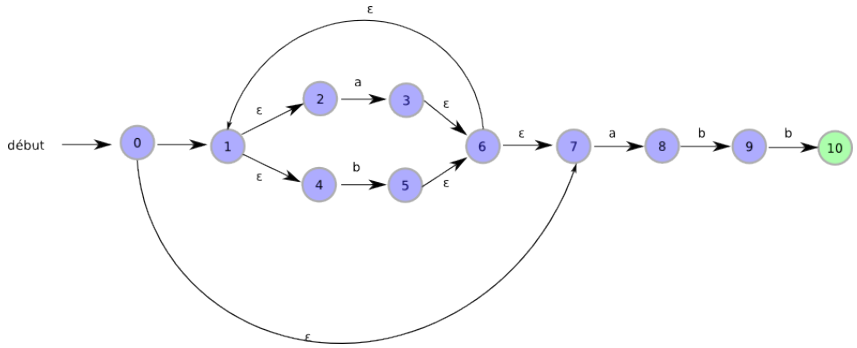


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(A,b))=ϵ-fermeture($\{5\}$)=$\{1, 2, 4, 5, 6, 7\}$=C

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B		
C		

$$\begin{aligned} A &= \{0, 1, 2, 4, 7\} \\ B &= \{1, 2, 3, 4, 6, 7, 8\} \\ C &= \{1, 2, 4, 5, 6, 7\} \end{aligned}$$

Exemple de transformation

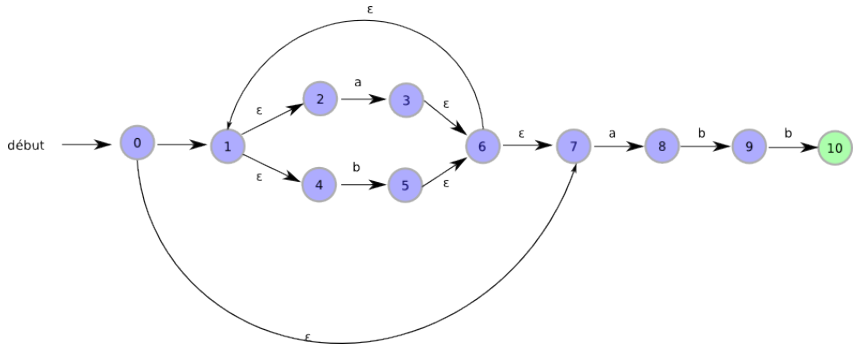


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(B,a))=ϵ-fermeture($\{3, 8\}$)=B

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B	B	
C		

$$\begin{aligned} A &= \{0, 1, 2, 4, 7\} \\ B &= \{1, 2, 3, 4, 6, 7, 8\} \\ C &= \{1, 2, 4, 5, 6, 7\} \end{aligned}$$

Exemple de transformation

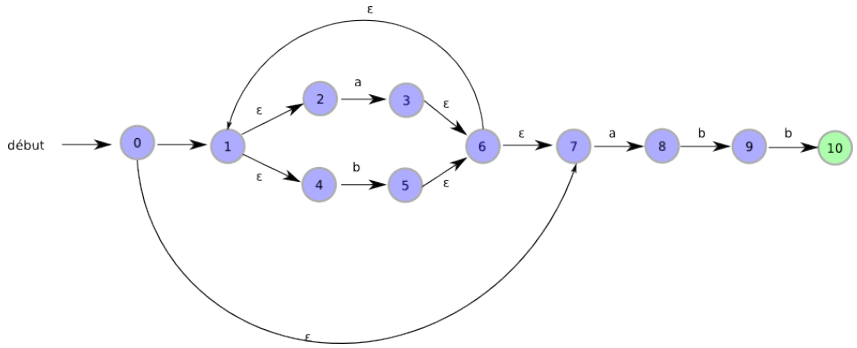


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(B,b))= ϵ-fermeture($\{5, 9\}$)=$\{1, 2, 4, 5, 6, 7, 9\}$=D

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B	B	D
C		
D		

$$
\begin{aligned}
A &= \{0, 1, 2, 4, 7\} \\
B &= \{1, 2, 3, 4, 6, 7, 8\} \\
C &= \{1, 2, 4, 5, 6, 7\} \\
D &= \{1, 2, 4, 5, 6, 7, 9\}
\end{aligned}
$$

Exemple de transformation

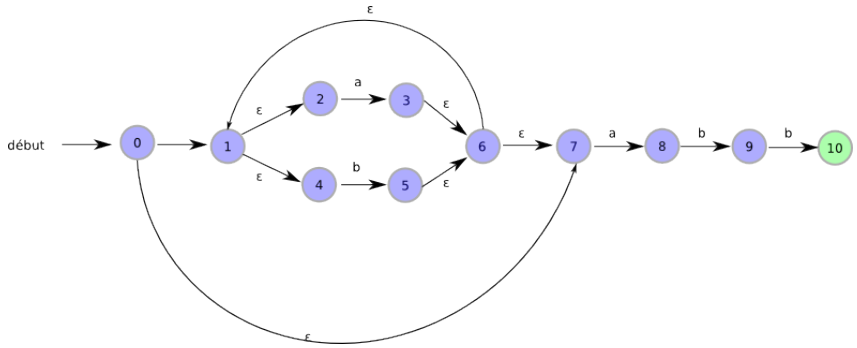


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(C,a))= ϵ-fermeture($\{3, 8\}$)=B

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B	B	D
C	B	
D		

$$
\begin{aligned}
A &= \{0, 1, 2, 4, 7\} \\
B &= \{1, 2, 3, 4, 6, 7, 8\} \\
C &= \{1, 2, 4, 5, 6, 7\} \\
D &= \{1, 2, 4, 5, 6, 7, 9\}
\end{aligned}
$$

Exemple de transformation

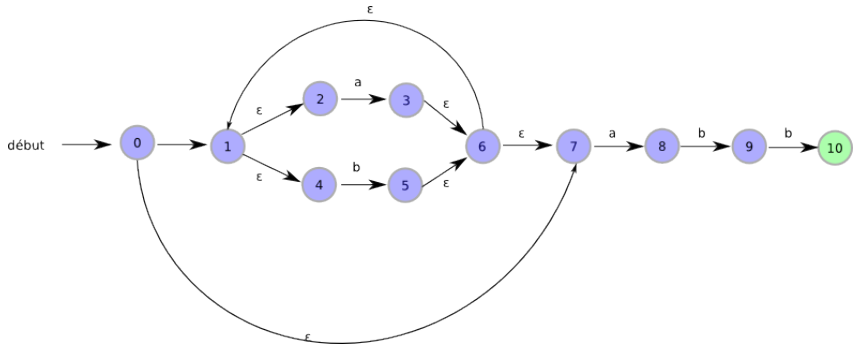


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(C,b))= ϵ-fermeture($\{5\}$)=C

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B	B	D
C	B	C
D		

$$
\begin{aligned}
A &= \{0, 1, 2, 4, 7\} \\
B &= \{1, 2, 3, 4, 6, 7, 8\} \\
C &= \{1, 2, 4, 5, 6, 7\} \\
D &= \{1, 2, 4, 5, 6, 7, 9\}
\end{aligned}
$$

Exemple de transformation

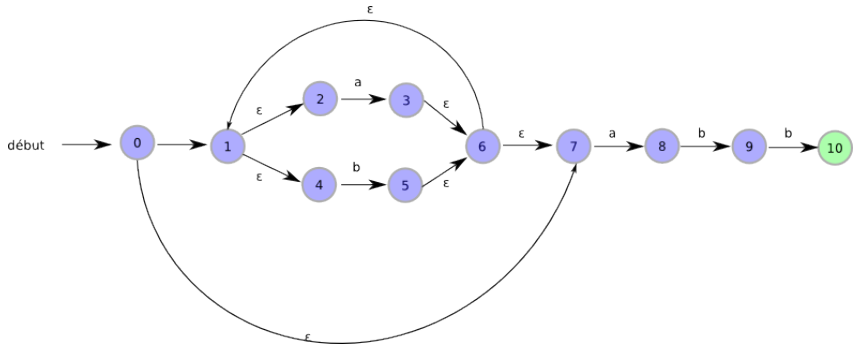


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(D,a))= ϵ-fermeture($\{3, 8\}$)=B

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B	B	D
C	B	C
D	B	

$$
\begin{aligned}
A &= \{0, 1, 2, 4, 7\} \\
B &= \{1, 2, 3, 4, 6, 7, 8\} \\
C &= \{1, 2, 4, 5, 6, 7\} \\
D &= \{1, 2, 4, 5, 6, 7, 9\}
\end{aligned}
$$

Exemple de transformation

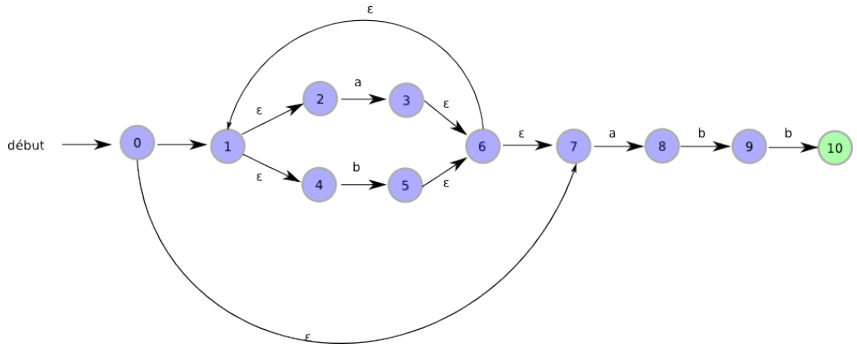


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(D,b))=
  ϵ-fermeture($\{5, 10\}$)=$\{1, 2, 4, 5, 6, 7, 10\}$=E

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E		

$$
\begin{aligned}
A &= \{0, 1, 2, 4, 7\} \\
B &= \{1, 2, 3, 4, 6, 7, 8\} \\
C &= \{1, 2, 4, 5, 6, 7\} \\
D &= \{1, 2, 4, 5, 6, 7, 9\} \\
E &= \{1, 2, 4, 5, 6, 7, 10\}
\end{aligned}
$$

Exemple de transformation

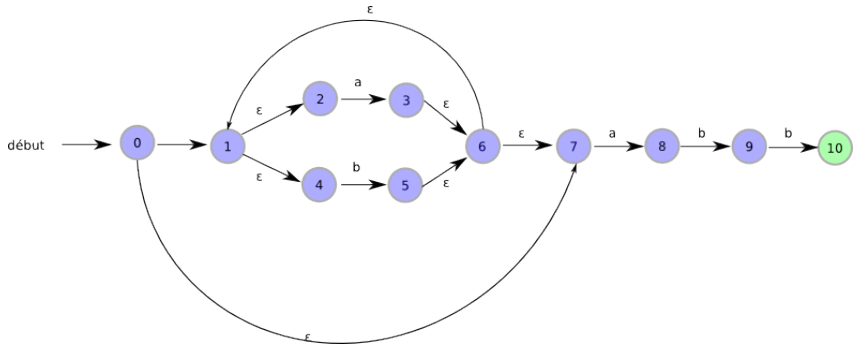


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(E,a))= ϵ-fermeture($\{3, 8\}$)=B

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	

$$
\begin{aligned}
A &= \{0, 1, 2, 4, 7\} \\
B &= \{1, 2, 3, 4, 6, 7, 8\} \\
C &= \{1, 2, 4, 5, 6, 7\} \\
D &= \{1, 2, 4, 5, 6, 7, 9\} \\
E &= \{1, 2, 4, 5, 6, 7, 10\}
\end{aligned}
$$

Exemple de transformation

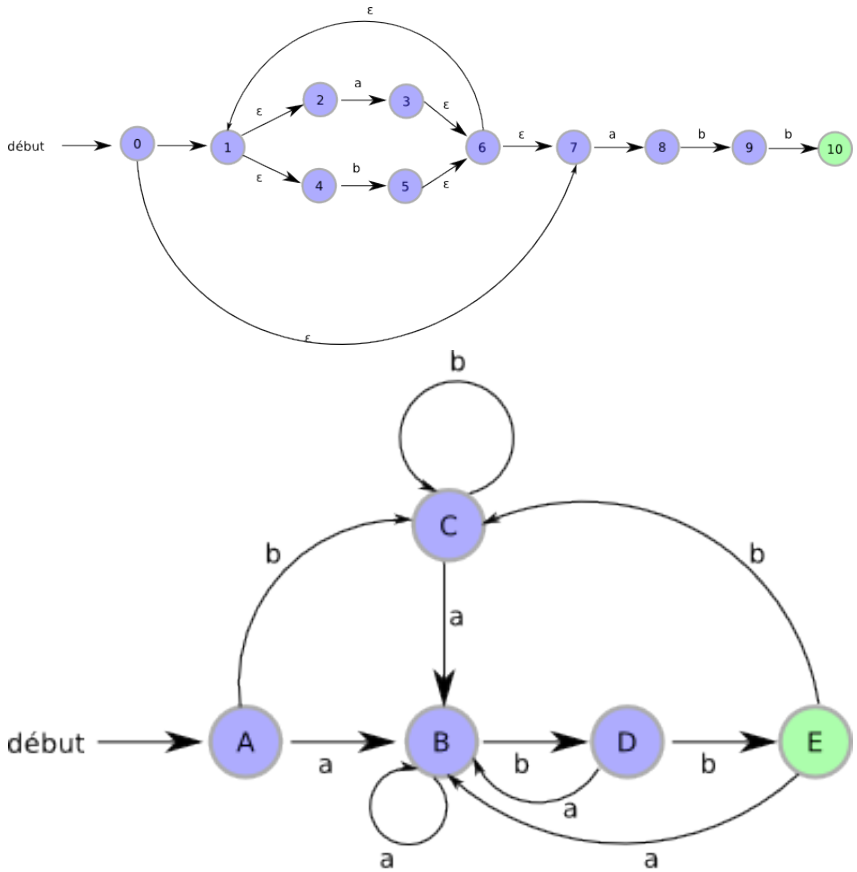


- ϵ-fermeture($\{0\}$)=$\{0, 1, 2, 4, 7\}$=A
- ϵ-fermeture(Transiter(E,b))= ϵ-fermeture($\{5\}$)=C

Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

$$
\begin{aligned}
A &= \{0, 1, 2, 4, 7\} \\
B &= \{1, 2, 3, 4, 6, 7, 8\} \\
C &= \{1, 2, 4, 5, 6, 7\} \\
D &= \{1, 2, 4, 5, 6, 7, 9\} \\
E &= \{1, 2, 4, 5, 6, 7, 10\}
\end{aligned}
$$

Exemple de transformation



Transiter		
Etat	Symbole d'entrée	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Discussion

- La reconnaissance d'un lexème par un AFD est plus rapide (une seule transition par état),
- En revanche le nombre d'états d'un AFD peut être exponentiel par rapport au nombres d'états de l'AFN.

Automate	Place	Temps
AFN	$\mathcal{O}(\|r\|)$	$\mathcal{O}(\|r\| * \|x\|)$
AFD	$\mathcal{O}(2^{\|r\|})$	$\mathcal{O}(\|x\|)$

- Différents compromis ou heuristiques (tel que l'évaluation paresseuse des transitions) sont donc utilisés.

Un outil d'analyse lexicale : Lex

- L'analyseur lexical Lex se combine avec l'analyseur syntaxique Yacc pour produire des compilateurs en C.
- Les cousins de Lex/Yacc pour le C++ se nomment Flex et Bisons.
- L'application de Lex sur un fichier produit un fichier `lex.yy.c` que l'on compile en utilisant la librairie Lex : libl.
- Ci joint un exemple de Makefile :

```
CC      = gcc
LEX     = lex
LEXFILE = monFichier.lex
OBJ     = lex.yy.o
OUTPUT  = nbId
LIB     = -ll

$(OUTPUT) :     $(OBJ)
                $(CC) $(OBJ) -o $@ $(LIB)

lex.yy.o :      lex.yy.c
                $(CC) -c lex.yy.c

lex.yy.c :      $(LEXFILE)
                $(LEX) $(LEXFILE)
```

Syntaxe des expressions régulières en Lex

x	le caractère "x"
.	n'importe quel caractère sauf \n
[xyz]	soit x, soit y, soit z
[^bz]	tous les caractères, sauf b et z
[a-z]	n'importe quel caractère entre a et z
[^a-z]	tous les caractères, sauf ceux compris entre a et z
r*	zéro r ou plus, où r est n'importe quelle expression régulière
r+	au moins un r
r ?	au plus un r (c'est-à-dire un r optionnel)
r{2,5}	entre 2 et 5 r
r{2,}	2 r ou plus
r{2}	exactement 2 r
rs	r suivi de s
r|s	r ou s
r/s	r, seulement s'il est suivi par s
^r	r, mais seulement en début de ligne
r$	r, mais seulement en fin de ligne

Syntaxe des expressions régulières en Lex

{r}	l'expansion de r
"[xyz\"foo"	la chaîne "[xyz"foo"
\X	si X est un "a", "b", "f", "n", "r", "t", ou "v", représente l'interprétation ANSI-C de \X.
\0	le caractère ASCII 0
\123	le caractère dont le code ASCII est 123, en octal
\x2A	le caractère dont le code ASCII est 2A, en hexadécimal
<<EOF>>	fin de fichier

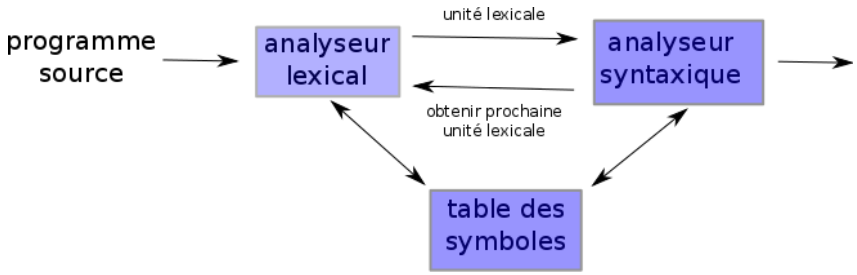
Structure d'un fichier Lex

%{
Déclaration des variables C
%}
Déclaration des Unités lexicales
%%
Déclarations des actions associées à la reconnaissance d'unités lexicales
%%
Code C supplémentaire (déclarations de fonctions auxilières)

Exemple de fichier Lex

```
%{
  int nb_numbers=0,nb_id=0;
%}
pm [+-]
chiffre [0-9]
frac \.{chiffre}+
exp E{pm}?{chiffre}+
nb {pm}?(({chiffre}+{frac}?)|{frac}){exp}?
lettre [a-zA-z]
id {lettre}({lettre}|{chiffre})*
%%
{nb}  { nb_numbers++;printf("Nombre %s reconnu : %f\n",yytext,atof(yyte
{id}  { nb_id++;printf("Id %s reconnu\n",yytext);}
{nb}{id} {printf("Lexical error\n");}
%%
int main()
{
yylex();
 printf("Nb Numbers: %d, \nNb Id %d\n",nb_numbers,nb_id);
return 0;
}
```

Analyse syntaxique



- L'analyseur lexical a en charge la reconnaissance des unités lexicales (nombres, mots clés, identifiants),
- L'analyseur syntaxique prend en charge l'agencement des unités lexicales pour former un langage.
- Les deux analyseurs collaborrent via la table des symboles.

Exemple :
Si $(x < 0)$ alors
x=-x
Finsi

$\longrightarrow$

Si Expr **alors**
Instr
Finsi

Analyse Syntaxique et grammaires

Nous allons utiliser les grammaires car :

- Les grammaires permettent une spécification facile et précise d'un langage,
- Pour les classes de grammaires usuellement utilisées, il existe des outils efficaces (Yacc, Bisons,...) permettant d'analyser automatiquement un fichier source à partir d'une grammaire.
- Les grammaires aident à spécifier proprement les langages de programmations ce qui est utile dans toutes les étapes ultérieures.
- Une spécification d'un langage par une grammaire permet de le faire évoluer (ajouter des fonctionnalités) simplement.

Grammaire non contextuelle

Une grammaire non contextuelle (/hors contexte/algébrique) est composée de productions :

$$X \rightarrow w$$

où X est appellé un non terminal (membre gauche de la règle) et w est une chaîne composée de terminaux et/ou de non terminaux.
Le terme non contextuel vient du fait que l'on remplace X par w sans référence au contexte où il apparaît.
Exemple de grammaire :

expr	→	*expr* *op* *expr*
expr	→	(*expr*)
expr	→	− *expr*
expr	→	**Id**
op	→	+
op	→	−
op	→	*
op	→	/

- *expr* et *op* sont des non terminaux
- **id**, +,-,*,/ sont des terminaux
- *expr* représente l'axiome : tout le langage est dérivé de l'axiome.

Notations

1. Terminaux : Les lettres minuscules du début de l'alphabet : a,b,c..., les symboles d'opérateurs +,-,..., les symboles de ponctuation '(',')',',',';'..., les chiffres 0 à 9 les chaînes en gras **id**, **Si** représentant des unités lexicales.
2. Non terminaux : Les lettres majuscules du début de l'alphabet : A,B,C..., la lettre S qui représente généralement l'axiome, les noms en italiques *expr* , *op*
3. Les lettres majuscules de la fin de l'alphabet X, Y, Z représentent des symboles grammaticaux (i.e. des terminaux ou non terminaux)
4. les chaînes minuscules de la fin de l'alphabet $u, v, w, \ldots, z$ représentent des chaînes de terminaux (ou phrases).
5. Les lettres greques minuscules α, β, γ représentent des chaînes de symboles grammaticaux ou proto phrases (composées de terminaux et non terminaux). Ex : $A \rightarrow \alpha$

Notations

6. La suite de productions $A \to \alpha_1, A \to \alpha_2, \ldots A \to \alpha_n$ se réécrit en $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$.
7. Sauf indication contraire la partie gauche de la première production indique l'axiome.

- La notation $E \Rightarrow -E$ signifie E se dérive en $-E$.
  Exemple :
  $$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(Id)$$
  De façon plus générale $\alpha A \beta \Rightarrow \alpha \gamma \beta$ si $A \to \gamma$.
- La notion se dérive en 0 à n étapes se note $\overset{*}{\Rightarrow}$. On a ainsi $E \overset{*}{\Rightarrow} -(Id)$ mais également $E \overset{*}{\Rightarrow} -(E)$.
- $\overset{+}{\Rightarrow}$ signifie de dérive en 1 à n étapes.

Vérification et expressions régulières

Le langage engendré par une grammaire est l'ensemble des phrases dérivées de son axiome. On dit dans ce cas que le langage est non contextuel.

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$$

- La preuve qu'une grammaire engendre un langage particulier se fait rarement sur des grammaires complètes mais peut se faire (et se fait) sur des parties de celles-ci ou des grammaires simples. On montre que $\forall w \in \Sigma^*$ tel que $S \overset{*}{\Rightarrow} w$ on a $w \in L(G)$ et inversement, pour tout $w \in L(G)$ il existe une séquence de réduction telle que $S \overset{*}{\Rightarrow} w$.
- Toute expression régulière peut être codée par une grammaire non contextuelle. Le contraire n'est évidement pas vrai. On restreint l'analyse lexicale à la reconnaissance des unités lexicales.

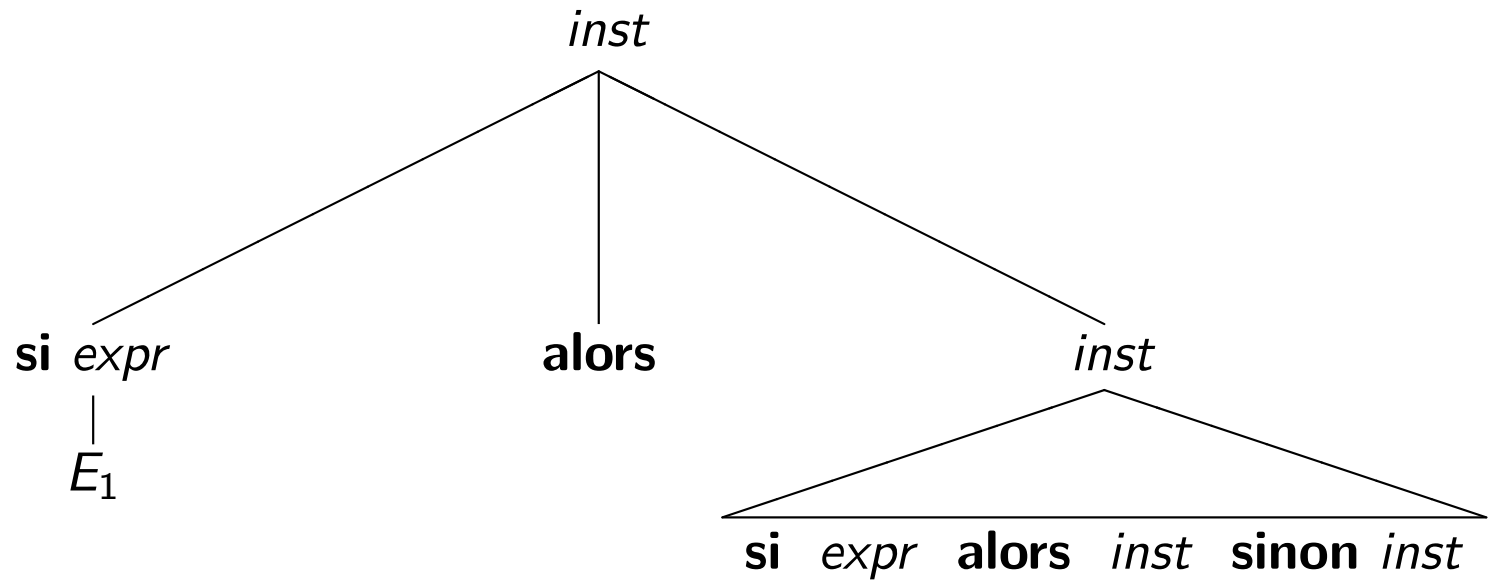
Suppression des ambiguïtés

La grammaire suivante :

$$
\begin{array}{lcl}
\textit{inst} & \rightarrow & \textbf{si}\ \textit{expr}\ \textbf{alors}\ \textit{inst} \\
 & & |\textbf{si}\ \textit{expr}\ \textbf{alors}\ \textit{inst}\ \textbf{sinon}\ \textit{inst} \\
 & & |\textbf{autre}
\end{array}
$$

produit une ambiguïté sur la proto-phrase :

si E_1 **alors si** E_2 **alors** S_1 **sinon** S_2



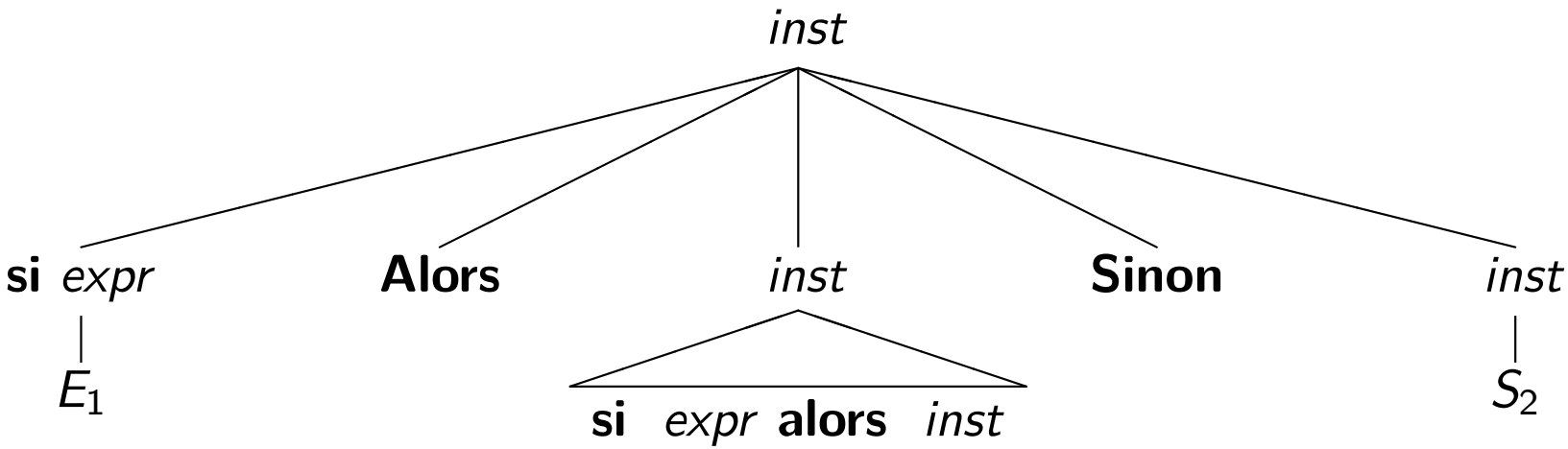
Suppression des ambiguïtés

La grammaire suivante :

$$
\begin{array}{lcl}
\textit{inst} & \rightarrow & \textbf{si}\ \textit{expr}\ \textbf{alors}\ \textit{inst} \\
 & & |\textbf{si}\ \textit{expr}\ \textbf{alors}\ \textit{inst}\ \textbf{sinon}\ \textit{inst} \\
 & & |\textbf{autre}
\end{array}
$$

produit une ambiguïté sur la proto-phrase :

si E_1 **alors si** E_2 **alors** S_1 **sinon** S_2



Suppression des ambiguïtés

Solution : Affecter chaque sinon au alors sans correspondant le plus proche. Réécrire la grammaire pour interdire les si alors sans correspondant dans des si alors sinon.

inst	$\rightarrow$	*instFermee*
		instNonFerme
instFermee	$\rightarrow$	**si** *expr* **alors** *instFermee* **sinon** *instFermee*
		autre
instNonFermee	$\rightarrow$	**si** *expr* **alors** *inst*
		|**si** *expr* **alors** *instFermee* **sinon** *instNonFermee*

On interdit les constructions du type :

$$\textbf{si } E_1 \textbf{ alors } (\textbf{si } E_2 \textbf{ alors } S_1) \textbf{ sinon } S_2$$

qui seront interprétées comme :

$$\textbf{si } E_1 \textbf{ alors } (\textbf{si } E_2 \textbf{ alors } S_1 \textbf{ sinon } S_2)$$

Suppression de la récursivité à gauche

Récusivité directe

Une grammaire récursive à gauche comporte productions telles que $A \overset{+}{\Rightarrow} A\alpha$. Ce genre de dérivation ne peuvent pas être analysés par des grammaires récursives gauches (celles que l'on étudie). On a donc besoins de transformer les productions $A \to A\alpha|\beta$ en les productions équivalentes :

$$\begin{array}{lcl} A & \to & \beta A' \\ A' & \to & \alpha A'|\epsilon \end{array}$$

Plus généralement si $A \to A\alpha_1| \ldots A\alpha_n|\beta_1| \ldots |\beta_m$, où A n'est un préfixe d'aucun β_i on remplace cette production par :

$$\begin{array}{lcl} A & \to & \beta_1 A'|\beta_2 A'| \ldots |\beta_m A' \\ A' & \to & \alpha_1 A'|\alpha_2 A'| \ldots |\alpha_n A'|\epsilon \end{array}$$

Suppression de la récursivité à gauche

Exemple :

$$
\begin{array}{lcl}
E & \rightarrow & E + T \,|\, T \\
T & \rightarrow & T * F \,|\, F \\
F & \rightarrow & (E) \,|\, \mathbf{id}
\end{array}
$$

se réécrit en :

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \,|\, \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \,|\, \epsilon \\
F & \rightarrow & (E) \,|\, \mathbf{id}
\end{array}
$$

Suppression de la récursivité à gauche :Récusivité indirecte

La récursivité à gauche peut être indirecte par exemple :

$$\begin{array}{lcl} S & \rightarrow & Aa|b \\ A & \rightarrow & Ac|Sd|\epsilon \end{array}$$

engendre la dérivation $S \overset{+}{\Rightarrow} Sda$.

- Idée : on ordonne les non terminaux et supprime itérativement les réduction permettant $A_i \Rightarrow A_j\alpha \Rightarrow A_i\beta\alpha$.
- Garantie de réussite uniquement si $A \not\overset{+}{\Rightarrow} A$ et si il n'y a pas de ϵ productions.

Suppression de la récursivité à gauche :Récusivité indirecte

La récursivité à gauche peut être indirecte par exemple :

$$\begin{array}{lcl} S & \rightarrow & Aa|b \\ A & \rightarrow & Ac|Sd|\epsilon \end{array}$$

engendre la dérivation $S \stackrel{+}{\Rightarrow} Sda$.

procédure suprRecGauche **(**E/S Grammaire S**)**
    Déclaration i,j : Entiers
début
    Ordoner les non terminaux $A_1, \ldots, A_n$
    pour i $\leftarrow$1 **à** n **faire**
        pour j $\leftarrow$1 **à** i-1 **faire**
            Remplacer $A_i \rightarrow A_j\gamma$ par $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\ldots|\delta_k\gamma$
            où $A_j \rightarrow \delta_1|\delta_2|\ldots|\delta_k$ sont les A_j production courantes
        finpour
        supprimer les récursivités gauches immédiates des A_i productions
    finpour
fin

Suppression de la récursivité à gauche :Récusivité indirecte

Exemple : Soit la grammaire suivante avec l'ordre S, A.

$$\begin{array}{lcl} S & \rightarrow & Aa|b \\ A & \rightarrow & Ac|Sd|\epsilon \end{array}$$

La première itération ne modifie pas $S \rightarrow Aa|b$. La seconde remplace $A \rightarrow Sd$ par :

$$A \rightarrow Aad|bd$$

qui s'ajoutent aux productions $A \rightarrow Ac|\epsilon$. On a donc $A \rightarrow Aad|Ac|bd|\epsilon$ dont on supprime la récursivité gauche pour obtenir :

$$\begin{array}{lcl} S & \rightarrow & Aa|b \\ A & \rightarrow & bdA'|A' \\ A' & \rightarrow & cA'|adA'|\epsilon \end{array}$$

Factorisation à gauche

Différentes productions peuvent avoir les mêmes préfixes. Par exemple :

$$
\begin{array}{lll}
\textit{inst} & \rightarrow & \textbf{si}\ \textit{expr}\ \textbf{alors}\ \textit{inst} \\
 & & |\textbf{si}\ \textit{expr}\ \textbf{alors}\ \textit{inst}\ \textbf{sinon}\ \textit{inst} \\
 & & |\textbf{autre}
\end{array}
$$

Sur rencontre de **si** *expr* **alors** *inst* laquelle des deux règles appliquer ?
On explicite ce choix par une factorisation gauche :

$$
\begin{array}{lll}
\textit{inst} & \rightarrow & \textbf{si}\ \textit{expr}\ \textbf{alors}\ \textit{inst}\ S' \\
S' & \rightarrow & \textbf{sinon}\ \textit{instr}\ |\epsilon
\end{array}
$$

Cette dernière grammaire est plus efficace car le choix ne s'effectue que quand il doit se faire.

Factorisation à gauche

De façon plus générale si il existe $\alpha \neq \epsilon$ tel que :

$$A \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma$$

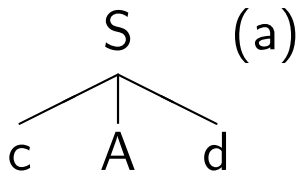
on remplace ces productions par :

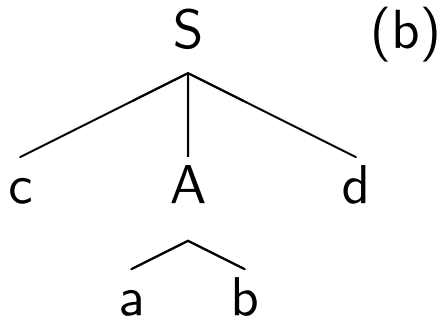
$$\begin{array}{lcl} A & \rightarrow & \alpha A'|\gamma \\ A' & \rightarrow & \beta_1|\beta_2|\ldots|\beta_n \end{array}$$

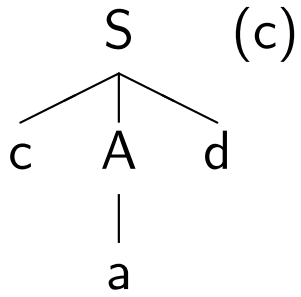
Le problème du rebroussement

Soit la grammaire :

$$S \rightarrow cAd$$
$$A \rightarrow ab|a$$







et la chaîne $w = cad$. On développe S(a). Le pointeur d'entré est positionné sur c. La feuille la plus a gauche étant égale à c nous avançons le pointeur d'entré sur a (second symbole de w) et appliquons la première production de A (b). On lit a sur le tampon d'entré et la feuille la plus à gauche, on avance le pointeur d'entré sur d et lisons la feuille b : Erreur. On backtraque donc et ramenons le pointeur sur c, on applique ensuite la deuxième production de A qui nous permet de reconnaître w.

Le problème du rebroussement

- On comprend sur l'exemple précédent que des grammaires récursives à gauche puissent faire boucler des analyseurs prédictifs.
- Notre objectif va être de supprimer autant que possible le backtrack quitte à restreindre les grammaires.

Grammaires et diagrammes de transitions

On associe un diagramme à chaque non terminal. Pour chaque production $A \to X_1 \ldots X_n$ en crée une chemin de l'état initial à l'état final avec les transitions $X_1 \ldots . X_n$.

$$
\begin{array}{lcl}
E & \to & TE' \\
E' & \to & +TE' | \epsilon \\
T & \to & FT' \\
T' & \to & *FT' | \epsilon \\
F & \to & (E) | \textbf{id}
\end{array}
$$

Exemple sur la grammaire d'expressions régulières simplifiées.

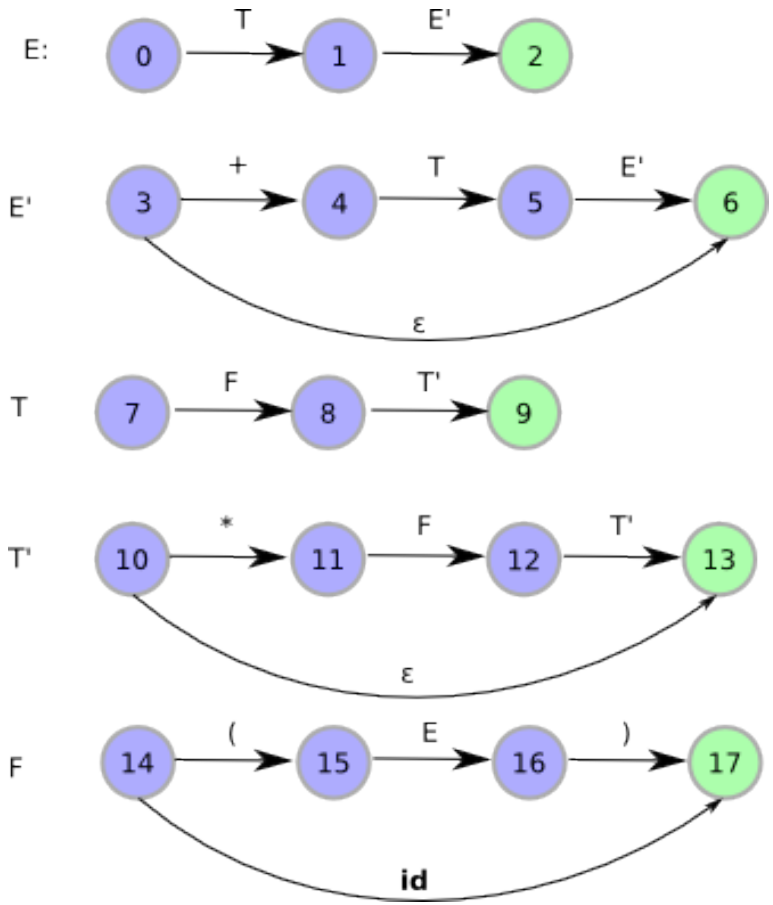
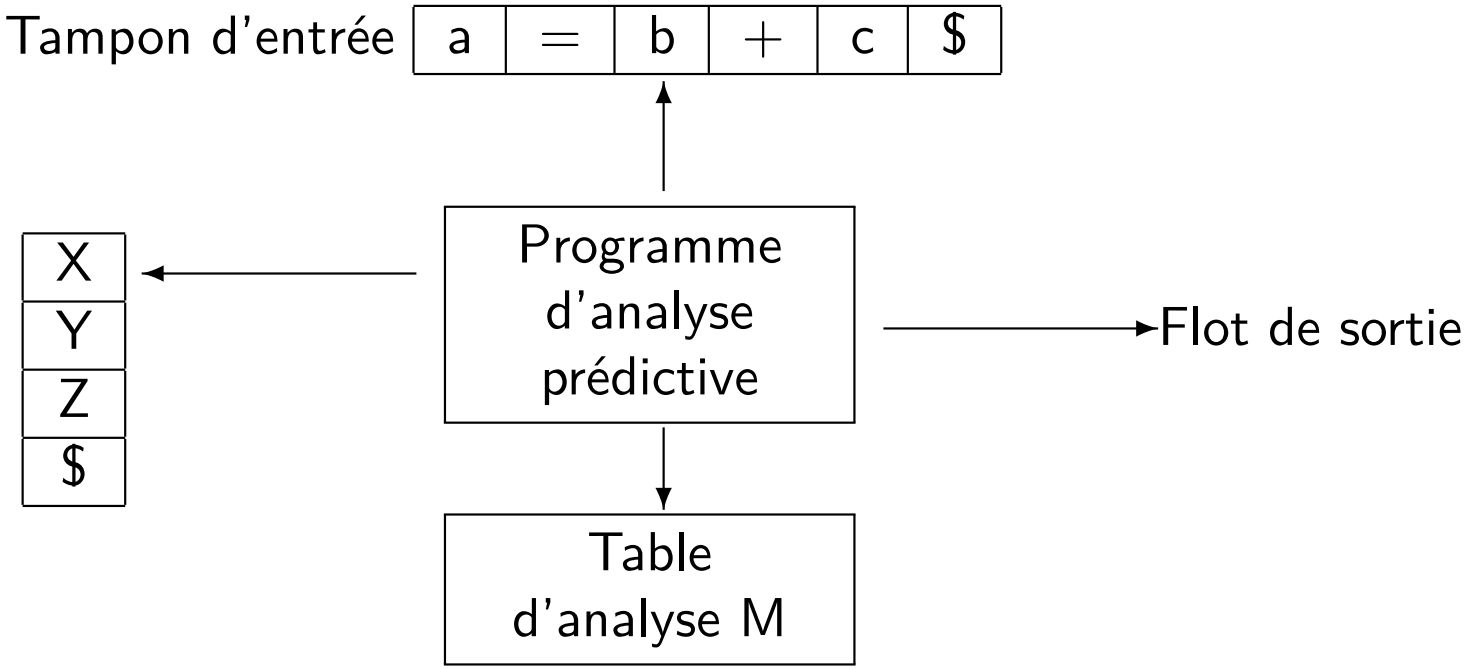


Schéma d'un analyseur syntaxtique prédictif

- Commencer dans l'état de départ de l'axiome.
- Si on est dans un état s avec une transition (s, t) étiquettée par le terminal a, et si le prochain symbole d'entré est a aller en t et décaller l'entrée d'un cran sur la droite.
- Si (s, t) est étiquetté par un non terminal A, aller dans l'état de départ de A. Si on atteint l'état final de A, passer à t.
- si (s, t) est étiquetté par ϵ on passe directement à t.

  On doit donc lire le tampon d'entré et empiler au moins les règles dans lequelles on rentre.

Analyse prédictive



L'analyseur lit le tampon d'entré et décide de la prochaine action à effectuer en fonction du sommet de pile et de la table M.

Analyse prédictive

Plus précisément : Si X est le sommet de pile et a l'entrée courante.

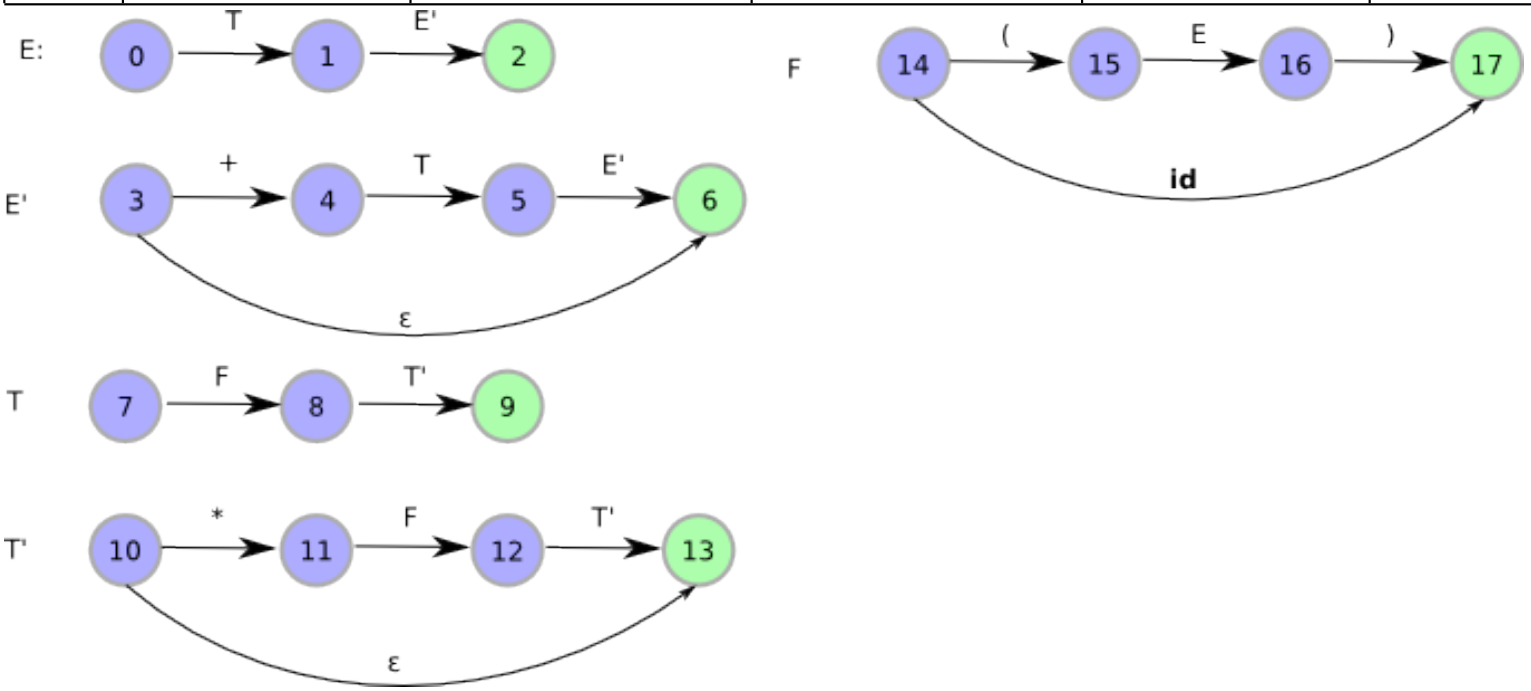
- Si $X = a = \$$ l'analyseur s'arrête et indique succès.
- Si $X = a \neq \$$ l'analyseur dépile X et décale le tampon d'entré.
- Si X est un non terminal. L'analyseur consulte $M[X, a]$. Si $M[X, a] = \textit{erreur}$, l'analyseur indique une erreur. Sinon $M[X, a]$ est une production $X \rightarrow UVW$ et l'analyseur dépile X, empile W, V et U (dans cet ordre)

Analyse prédictive : Algo

procédure AnalysePredictive **(**E P : pile, M : table, T : tampon**)**
    Déclaration ps : pointeur sur T
début
    répéter
        X $\leftarrow$ sommet de P
        a $\leftarrow$ valeur pointée par ps
        si X est un terminal ou $ **alors**
            si X=a **alors** dépiler X, avancer ps **sinon** Erreur()
        sinon
            si M[X,a]=$X \rightarrow Y_1 \dots Y_n$ **alors**
                dépiler X, Empiler $Y_n, \dots, Y_1$
                Emmetre en sortie $X \rightarrow Y_1 \dots Y_n$
            sinon
                Erreur()
            finsi
        finsi
    jusqu'à ce que $X = \$$
fin

Exemple de table d'analyse

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		



PREMIER et SUIVANT

Une question récurente en compilation est : L'appliquation de cette production me permettra elle de décaler tel symbole sur le tampon d'entré. Les fonctions PREMIER et SUIVANT nous aident à y répondre.

- Si α est une proto phrase, PREMIER(α) est l'ensemble des terminaux a tels que $\alpha \stackrel{*}{\Rightarrow} aw$. Si $\alpha \stackrel{*}{\Rightarrow} \epsilon$ alors $\epsilon \in PREMIER(\alpha)$.
- Si A est un non terminal SUIVANT(A) est l'ensemble des terminaux a tels qu'il existe une dérivation $S \stackrel{*}{\Rightarrow} \alpha A a \beta$. Si A est le symbole le plus a droite d'une proto-phrase ($S \stackrel{*}{\Rightarrow} \alpha A$) alors \$ $\in SUIVANT(A)$.

Calcul de PREMIER

Calcul de PREMIER(X), pour tout symbole X de la grammaire

1. Si X est un terminal, $PREMIER(X) = \{X\}$
2. Si $X \rightarrow \epsilon$, ajouter ϵ à PREMIER(X)
3. Si X est un non terminal et $X \rightarrow Y_1 \dots Y_k$.

procédure UpdatePremier **(**PREMIER(X), $Y_1, \dots Y_k$**)**
début
    i $\leftarrow$ 1
    répéter
        PREMIER(X) $\leftarrow PREMIER(X) \cup PREMIER(Y_i)$
        i $\leftarrow$ i+1
    jusqu'à ce que $\epsilon \notin PREMIER(Y_{i-1})$
fin

Le calcul de PREMIER($X_1 \dots X_n$) se fait par une méthode analogue à UpdatePremier.

Calcul de SUIVANT

1. Ajouter $ à PREMIER(S), S l'axiome, $ la fin de chaîne.
2. Pour toute production $B \to \alpha A \beta$,

$$SUIVANT(A) = SUIVANT(A) \cup (PREMIER(\beta) - \{\epsilon\}) .$$

3. Pour toute production $B \to \alpha A$ ou $B \to \alpha A \beta$ avec $\epsilon \in PREMIER(\beta)$,

$$SUIVANT(A) = SUIVANT(A) \cup SUIVANT(B)$$

Calcul de PREMIER et SUIVANT : Exemple

Reprenons notre grammaire :

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE'|\epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT'|\epsilon \\
F & \rightarrow & (E)|\mathbf{id}
\end{array}
$$

PREMIER(F)	=	{(,id}	(3)	SUIVANT(E)	=	{$,)}(1&2)
PREMIER(T')	=	{*,ϵ}	(2 & 3)	SUIVANT(E')	=	{$,)} (3)
PREMIER(T)	=	{(,id}	(3)	SUIVANT(T)	=	{+,$,)} (2&3)
PREMIER(E')	=	{+,ϵ}	(2 & 3)	SUIVANT(T')	=	{+,$,)} (3)
PREMIER(E)	=	{(,id}	(3)	SUIVANT(F)	=	{+,$,),*} (2&3

Construction des tables

- La production $A \rightarrow \alpha$, peut être utilisée si l'entré sur le tampon d'entré appartient à PREMIER(α).
- Si $\alpha \stackrel{*}{\Rightarrow} \epsilon$, on peut toujours utiliser cette règle si le symbole d'entré appartient à SUIVANT(A) (y compris si ce symbole est \$).

Donc on construit la table de la façon suivante :

1. Pour chaque production $A \rightarrow \alpha$ faire les étapes 2 et 3.
2. Pour chaque terminal a dans PREMIER(α), ajouter $A \rightarrow \alpha$ à M[A,a]
3. Si $\epsilon \in \mathit{PREMIER}(\alpha)$, ajouter $A \rightarrow \alpha$ à M[A,b] pour tout $b \in \mathit{SUIVANT}(A)$. Si $\epsilon \in \mathit{PREMIER}(\alpha)$ et \$ $\in \mathit{SUIVANT}(A)$ ajouter $A \rightarrow \alpha$ à M[A,\$].
4. Toute entrée de M non remplie correspond à une erreur.

Grammaires LL(1)

Toute grammaire dont la table remplie par l'algorithme précédent ne comporte pas d'entrée multiple est appellée **LL(1)** : **L**eft to right scanning, **L**eftmost derivation, utilisant **1** symbole de prévision.

Une grammaire est LL(1) ssi pour toute production $A \rightarrow \alpha|\beta$

1. Pour aucun terminal *a* nous avons $\alpha \stackrel{*}{\Rightarrow} aw$ et $\beta \stackrel{*}{\Rightarrow} aw'$,
2. Au plus une des chaîne α ou β peut se dériver en la chaîne vide,
3. Si $\beta \stackrel{*}{\Rightarrow} \epsilon$, α ne se dérive pas en une chaîne commençant par un terminal de SUIVANT(A).

Notre grammaire des expression arithmétiques est LL(1). La grammaire correspondant au si sinon ne l'est pas.

Grammaires LL(1) : Conclusions

-  Les grammaires LL(1) restent suffisament simples pour pouvoir être développées « à la main ».
-  Le pouvoir expressif de ces grammaires reste très limité.

Analyse ascendante

- A l'inverse des méthodes descendantes qui sont basées sur une dérivation de l'axiome les méthodes d'analyse par décalage-réduction sont des méthodes ascendantes qui partent de la chaîne (des feuilles) pour remonter à l'axiome.
- A chaque étape la partie droite d'une production présente dans la chaîne est remplacée par sa partie gauche.
- Si on fait les bons choix (et si la chaîne appartient au langage) on remonte ainsi à l'axiome.

Analyse ascendante : exemple

Considérons la grammaire :

$$\begin{array}{lcl} S & \rightarrow & aABe \\ A & \rightarrow & Abc|b \\ B & \rightarrow & d \end{array}$$

Et $w = abbcde$, le terminal le plus à gauche, b peut être remplacé par A pour obtenir $aAbcde$. $aAbcde$ peut être réécrit en $aAde$, puis en $aABe$ et finalement en S. Cela correspond à la séquence de dérivations droite suivante :

$$S \underset{d}{\Rightarrow} aABe \underset{d}{\Rightarrow} aAde \underset{d}{\Rightarrow} aAbcde \underset{d}{\Rightarrow} abbcde$$

$\underset{d}{\Rightarrow}$ signifie que l'on applique une réduction sur le terminal le plus à droite du mot.
Notez également que l'utilisation de la production $A \rightarrow b$ sur $aAbcde$ aurait donné $aAAcde$ ce qui ne permettait pas de revenir à S.

Manche

Si, il existe $A \to \beta$ tel que :

$$S \underset{d}{\overset{*}{\Rightarrow}} \alpha A w \underset{d}{\Rightarrow} \alpha\beta w, \qquad w \in \Sigma^*$$

Le couple $(A \to \beta, |\alpha| + 1)$ est appelé un manche de $\alpha\beta w$.
Notez que :

- La chaîne w qui suit β est composée uniquement de terminaux.
- Un manche est identifié à la fois par sa production et par la position de son membre droit dans la chaîne.
- Un manche n'existe qu'à partir d'une dérivation de l'axiome. Donc, donné une phrase γ identifier β dans γ tel que $\gamma = \alpha\beta w, w \in \Sigma^*$ ne suffit pas. Il faut également que $S \overset{*}{\Rightarrow} \gamma$.

Intuitivement, un manche est un mot que l'on va saisir pour remonter une dérivation.

Manche : Exemple

Considérons :

$$\begin{array}{lcl} S & \rightarrow & aABe \\ A & \rightarrow & Abc|b \\ B & \rightarrow & d \end{array}$$

et :

$$S \underset{d}{\Rightarrow} aABe \underset{d}{\Rightarrow} aAde \underset{d}{\Rightarrow} aAbcde \underset{d}{\Rightarrow} abbcde$$

- $(A \rightarrow b, 2)$ est un manche de $abbcde$ $(\alpha = a, \beta = b, w = bcde)$
- $(A \rightarrow Abc, 2)$ est un manche de $aAbcde$
- $(B \rightarrow d, 3)$ est un manche de $aAde$
- $(S \rightarrow aABe, 1)$ est un manche de $aABe$

Élagage du manche

Partant d'un mot composé de terminaux $w = \gamma_n$, on trouve un manche $(A_n \to \beta_n, i_n)$ et on remplace dans γ_n, β_n en position i_n par A_n pour obtenir γ_{n-1}, on cherche à nouveau le manche de γ_{n-1} pour obtenir γ_{n-2} et ainsi de suite jusqu'à $\gamma_0 = S$. On a donc :

$$S = \gamma_0 \underset{d}{\Rightarrow} \gamma_1 \underset{d}{\Rightarrow} \gamma_2 \cdots \underset{d}{\Rightarrow} \gamma_{n-1} \underset{d}{\Rightarrow} \gamma_n = w$$

Élagage du manche : Exemple

Soit la grammaire

$$\begin{aligned} E &\rightarrow E + E | E * E | (E) \\ E &\rightarrow id \end{aligned}$$

et $w = id_1 + id_2 * id_3$. On obtient la séquence de réduction suivante :

Proto phrase droite	Manche	Production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Notons que nous avons 2 manches possibles à la ligne 3 ($E \rightarrow E + E$ et $E \rightarrow id$) que l'on résout en utilisant la priorité de la multiplication.

Analyse par décalage-réduction avec une pile

L'analyseur est constitué d'une pile et d'un tampon d'entré. A chaque étape, l'extrémité droite du manche (si il existe) se trouve en sommet de pile. L'analyseur ajoute un $ en fin de la chaîne à reconnaître et en début de pile puis effectue les actions suivantes :

1. Si on ne trouve pas de manche dans la pile, on effectue une action *décaler* : On déplace le prochain symbole du tampon d'entré dans la pile.
2. Si il existe un manche $(A \rightarrow \beta)$ en sommet de pile, on effectue une action *réduire* : la séquence de symboles définissant β est remplacée par A dans la pile.
3. Si la pile contient $\$S$, S l'axiome et le tampon d'entré $ l'analyseur effectue une action *accepter* : La chaîne d'entré est reconnue.
4. Si aucun manche ne peut être trouvé par une action décaler, l'analyseur effectue une action *erreur*.

Analyse par décalage-réduction : Exemple

Pile	Entrée	Action
(1) \$	$id_1 + id_2 * id_3\$$	décaler
(2) \$$id_1$	$+id_2 * id_3\$$	réduire par $E \rightarrow Id$
(3) \$E	$+id_2 * id_3\$$	décaler
(4) \$E+	$id_2 * id_3\$$	décaler
(5) \$E+id_2	$*id_3\$$	réduire par $E \rightarrow Id$
(6) \$E+E	$*id_3\$$	décaler
(7) \$E+E*	$id_3\$$	décaler
(8) \$E+E*id_3	\$	réduire par $E \rightarrow Id$
(9) \$E+E*E	\$	réduire par $E \rightarrow E * E$
(10) \$E+E	\$	réduire par $E \rightarrow E + E$
(11) \$E	\$	accepter

Notez le conflit décaler/réduire ligne 6 résolu en utilisant la priorité des opérateurs. Il existe en fait 2 types de conflit : décaler/réduire et réduire/réduire quand il existe des non terminaux A_1, A_2 tels que $A_1 \rightarrow \beta$ et $A_2 \rightarrow \beta$.

Préfixes viables

- Un préfixe viable est un préfixe d'une proto phrase ne s'étendant pas au delà de la fin du manche le plus a droite de la proto phrase.
- Par exemple : $E + E$ est un préfixe viable de $E + E * id_3$
- Dans l'algorithme précédent, on peut donc décaler des symboles du tampon d'entré tant que le contenu de la pile (lu de la base au sommet) constitue un préfixe viable de la proto-phrase constitué de la pile et du tampon d'entré.

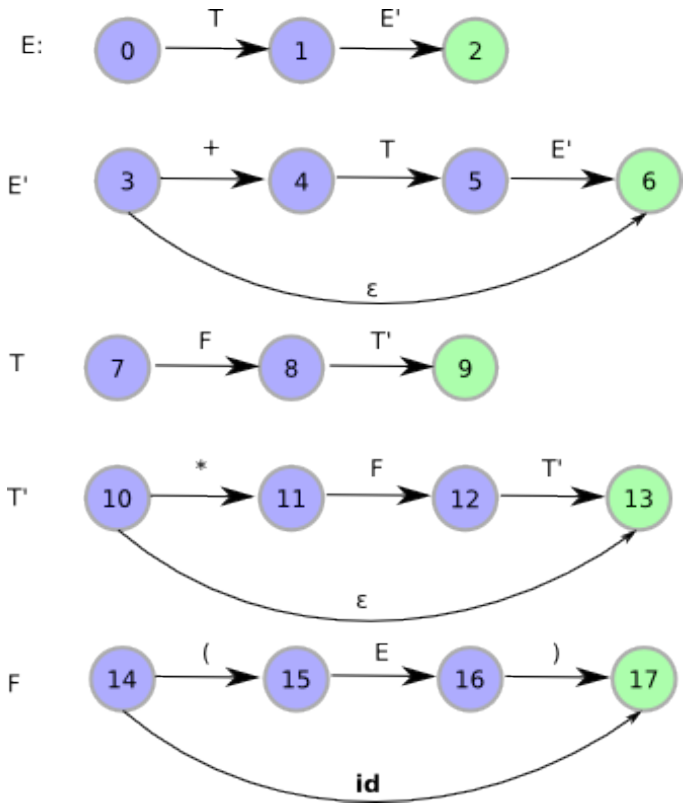
Analyseurs LR

LR(k)	**L**eft to right scanning of the input constructing a **R**ightmost derivation in reverse using **k** symbols.

-  Tous les langages que vous avez vu, peuvent être reconnus par un analyseur LR.
-  La méthode d'analyse LR est la méthode par décalage réduction sans rebroussement la plus générale connue. Elle peut cependant être implantée aussi efficacement que d'autre méthodes utilisant le même principe.
-  L'ensemble des grammaires reconnaissables par un analyseur LR est un sur ensemble strict des grammaires pouvant être reconnue par un analyseur prédictif.
-  Les analyseurs LR peuvent détecter très tôt les erreurs de syntaxe.
-  les tables d'analyse LR sont trop grandes pour être rédigées « à la main ». Heureusement de très bon outils (Yacc, Bison) existent.

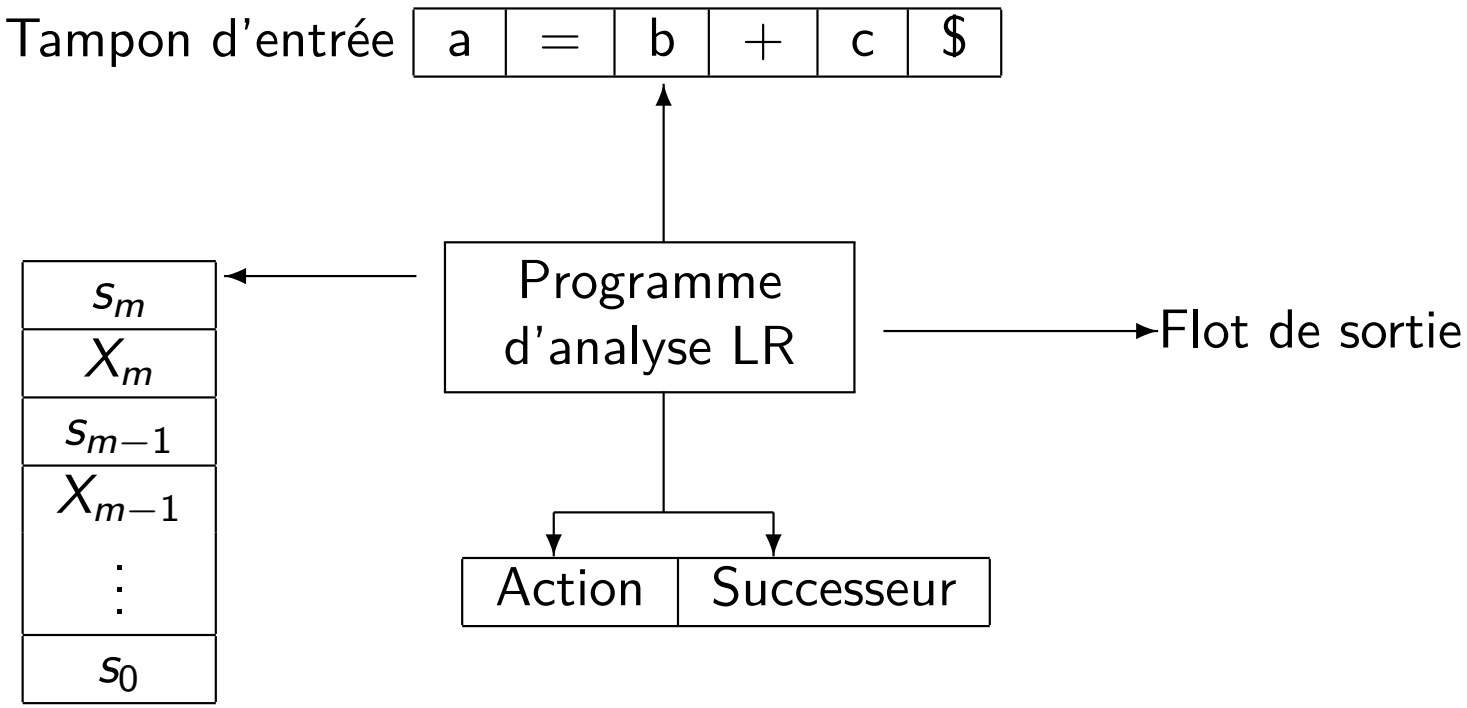
Analyse LR : Idée de base

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE'|\epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT'|\epsilon \\
F & \rightarrow & (E)|\mathbf{id}
\end{array}
$$



- Les algorithmes d'analyse précédent prennent une décision sur la vue de T en sommet de pile et la lecture d'un symbole dans le tampon d'entré.
- Voit on T depuis l'état 0 ou depuis l'état 4 ?
- Les méthodes d'analyse précédentes (dont celle des langages LL(1)) manquent de « mémoire ».

Analyse LR : schéma



Analyse LR : schéma

- La pile contient une séquence $s_0 X_1 \dots X_m s_m$ alternant des états s_i et des symboles X_i représentant les transitions entre ces états.
- L'état s_m résume donc l'état de la pile.
- Si a est le symbole courant du tampont s'entré et s_m le sommet de la pile Action$[s_m, a]$ correspond à une des actions suivantes :
  - décaler s, où s est un état,
  - réduire par une production $A \to \beta$
  - accepter
  - erreur
- Successeur[s,X] (s un état, X un non terminal) renvoie un état.

Analyse LR : Algorithme

Une configuration de l'analyseur est donné par le couple (pile,tampon d'entré) : $(s_0 X_1 \dots X_m s_m, a_i \dots, a_n\$)$ représentant la proto-phrase : $X_1 X_2 \dots X_{m-1} X_m a_i \dots, a_n$.

1. Si Action$[s_m, a_i]$= décaler s. On décale a_i et empile s pour passer dans la configuration :
$$(s_0 X_1 \dots X_m s_m a_i s, a_{i+1} \dots, a_n\$)$$
2. Si Action$[s_m, a_i]$=réduire par $A \to \beta = X_{m-r+1} \dots X_m$. On passe dans la configuration :
$$(s_0 X_1 \dots s_{m-r} A\ s, a_i \dots, a_n\$)$$
avec s=Sucesseur$[s_{m-r}, A]$. Le tampon d'entré reste inchangé.
3. Si Action$[s_m, a_i]$=accepter : renvoyer succès
4. Si Action$[s_m, a_i]$=erreur, appeler une routine de récupération d'erreur.

Analyse LR : Algorithme

fonction analyseLR **(**w$: chaîne**) :** Booleen
    Déclaration a : symbole courrant du tampon , s : Sommet de pile
début
    répéter
        si Action[s,a]= décaler s' **alors**
            empiler a puis s'
            avancer sur le prochain symbole du tampon d'entré
        sinon
            si Action[s,a]= réduire par $A \rightarrow \beta$ **alors**
                dépiler $2|\beta|$ symboles
                s' $\leftarrow$ nouveau sommet de pile
                empiler A puis Successeur[s',A]
            finsi
            si Action[s,a]=Erreur **alors** Erreur()
        finsi
    jusqu'à ce que Action[s,a]=accepter
fin

Analyse LR : Exemple

Toutes les méthodes d'analyse LR sont basées sur l'algorithme précédent. La différence intervient sur les méthodes d'initialisations des tables Actions et Successeur.
Considérons la grammaire :

$$
\begin{array}{lrcl}
(1) & E & \rightarrow & E + T \\
(2) & E & \rightarrow & T \\
(3) & T & \rightarrow & T * F \\
(4) & T & \rightarrow & F \\
(5) & F & \rightarrow & (E) \\
(6) & F & \rightarrow & \textbf{id}
\end{array}
$$

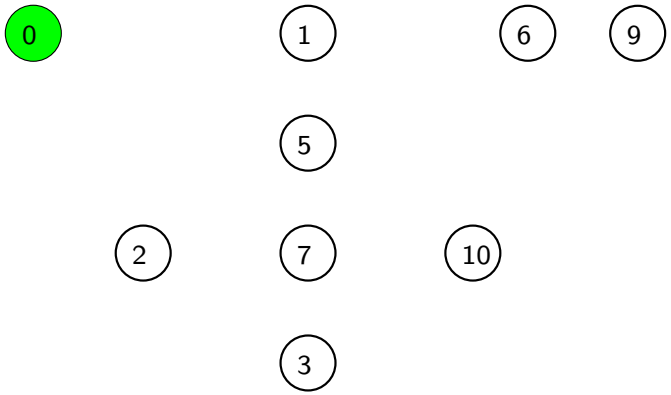
avec les conventions suivantes :

- di : décaler et empiler i
- rj : réduire par la production j
- acc : accepter
- entré vide : erreur

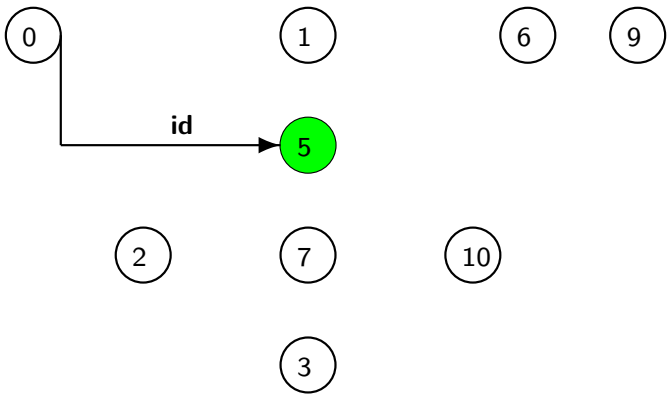
Analyse LR : Exemple

On obtient les tables Actions/ successeur suivantes (nous verrons plus loin comment les calculer).

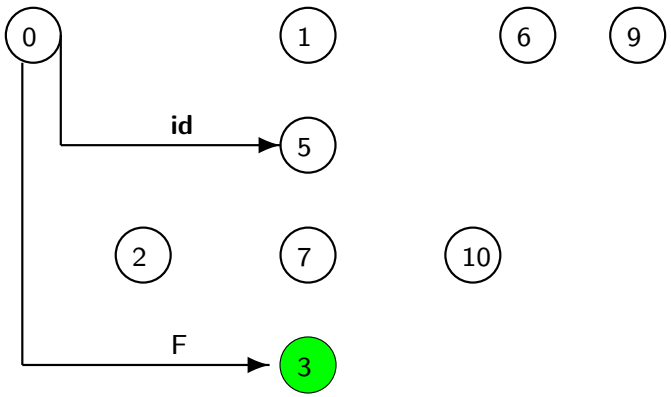
État	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r4	r4			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			



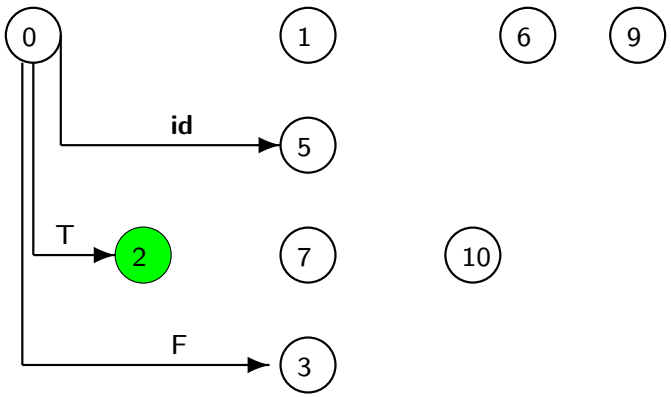
Pile	Entrée	Action
(1) 0	**id*id+id**	d 5



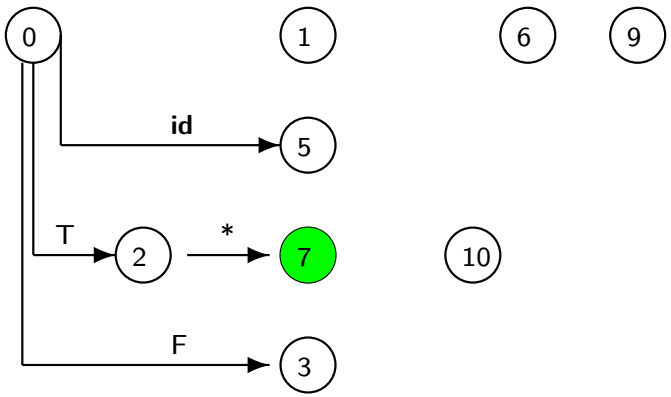
Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**



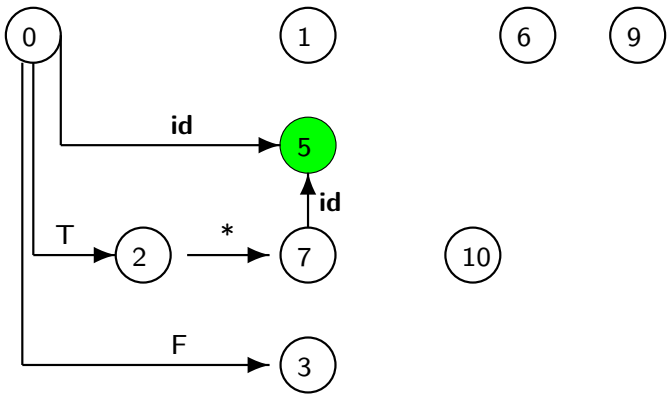
Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**\$	r par $T \rightarrow F$



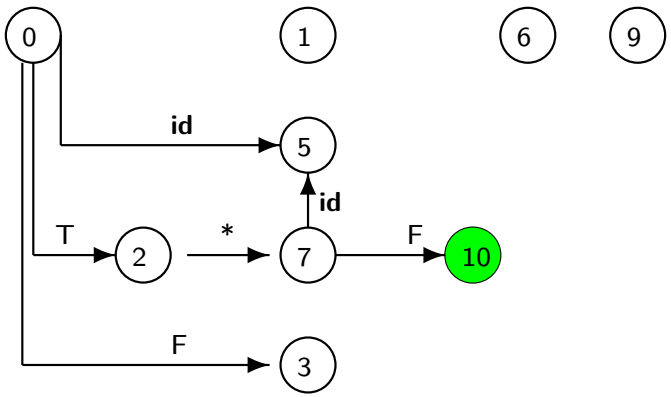
Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**$	d 7



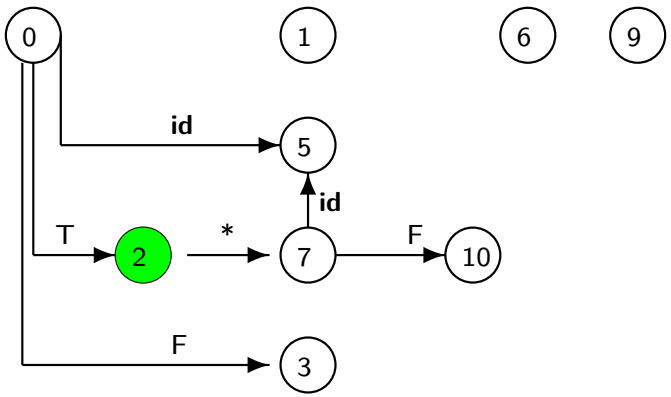
Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**\$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**\$	d 7
(5) 0 T 2 * 7	**id**+**id**\$	d 5



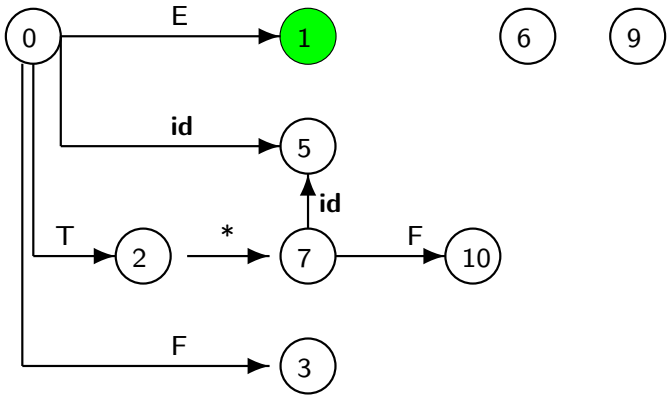
Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**\$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**\$	d 7
(5) 0 T 2 * 7	**id**+**id**\$	d 5
(6) 0 T 2 * 7 **id** 5	+**id**\$	r par $F \rightarrow$ **id**



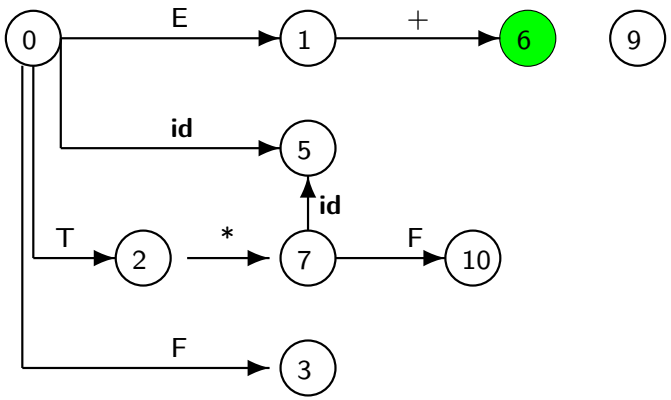
Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**\$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**\$	d 7
(5) 0 T 2 * 7	**id**+**id**\$	d 5
(6) 0 T 2 * 7 **id** 5	+**id**\$	r par $F \rightarrow$ **id**
(7) 0 T 2 * 7 F 10	+**id**\$	r $T \rightarrow T * F$



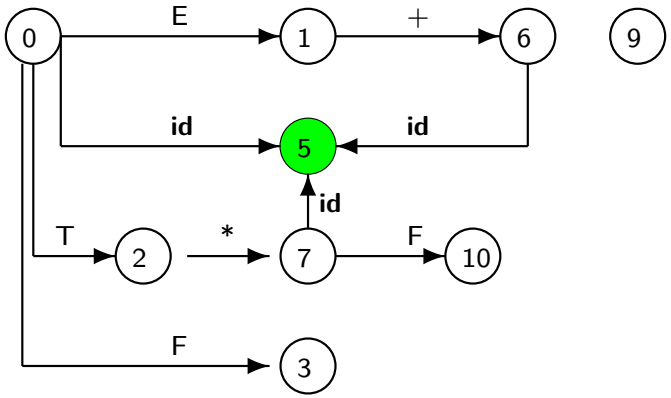
Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**\$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**\$	d 7
(5) 0 T 2 * 7	**id**+**id**\$	d 5
(6) 0 T 2 * 7 **id** 5	+**id**\$	r par $F \rightarrow$ **id**
(7) 0 T 2 * 7 F 10	+**id**\$	r $T \rightarrow T * F$
(8) 0 T 2	+**id**\$	r $E \rightarrow T$



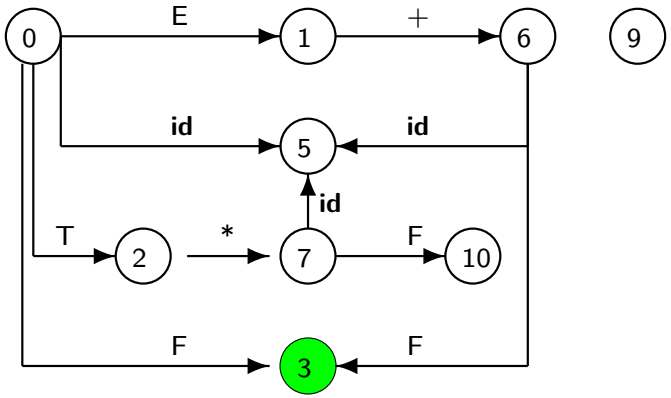
Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**\$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**\$	d 7
(5) 0 T 2 * 7	**id**+**id**\$	d 5
(6) 0 T 2 * 7 **id** 5	+**id**\$	r par $F \rightarrow$ **id**
(7) 0 T 2 * 7 F 10	+**id**\$	r $T \rightarrow T * F$
(8) 0 T 2	+**id**\$	r $E \rightarrow T$
(9) 0 E 1	+**id**\$	d 6



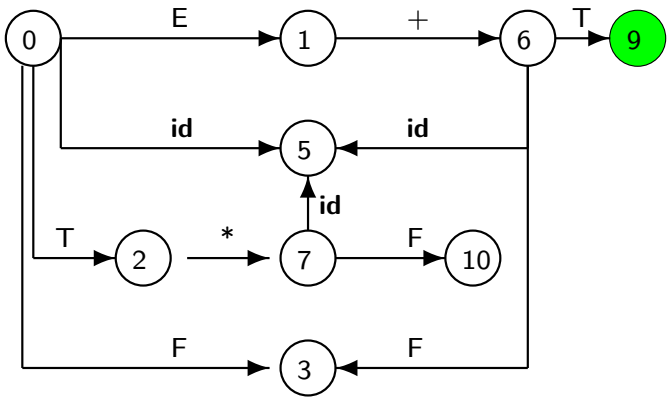
Pile	Entrée	Action
(1) 0	**id*id+id**	d 5
(2) 0 **id** 5	***id+id$**	r par $F \rightarrow$ **id**
(3) 0 F 3	***id+id$**	r par $T \rightarrow F$
(4) 0 T 2	***id+id$**	d 7
(5) 0 T 2 * 7	**id+id$**	d 5
(6) 0 T 2 * 7 **id** 5	**+id$**	r par $F \rightarrow$ **id**
(7) 0 T 2 * 7 F 10	**+id$**	r $T \rightarrow T * F$
(8) 0 T 2	**+id$**	r $E \rightarrow T$
(9) 0 E 1	**+id$**	d 6
(10) 0 E 1 + 6	**id$**	d 5



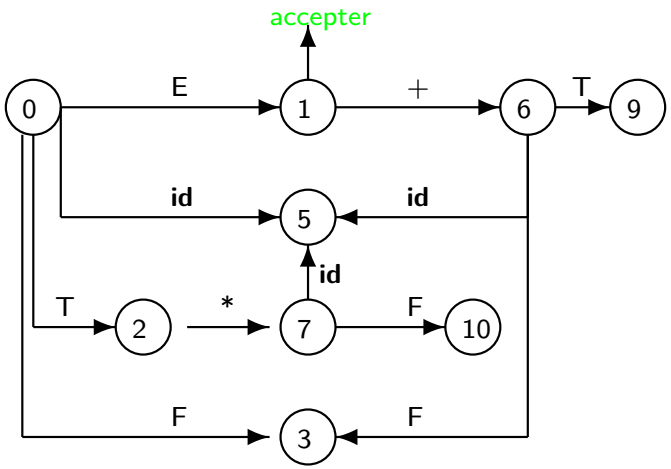
Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**$	d 7
(5) 0 T 2 * 7	**id**+**id**$	d 5
(6) 0 T 2 * 7 **id** 5	+**id**$	r par $F \rightarrow$ **id**
(7) 0 T 2 * 7 F 10	+**id**$	r $T \rightarrow T * F$
(8) 0 T 2	+**id**$	r $E \rightarrow T$
(9) 0 E 1	+**id**$	d 6
(10) 0 E 1 + 6	**id**$	d 5
(11) 0 E 1 + 6 **id** 5	$	r par $F \rightarrow$ **id**



Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**\$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**\$	d 7
(5) 0 T 2 * 7	**id**+**id**\$	d 5
(6) 0 T 2 * 7 **id** 5	+**id**\$	r par $F \rightarrow$ **id**
(7) 0 T 2 * 7 F 10	+**id**\$	r $T \rightarrow T * F$
(8) 0 T 2	+**id**\$	r $E \rightarrow T$
(9) 0 E 1	+**id**\$	d 6
(10) 0 E 1 + 6	**id**\$	d 5
(11) 0 E 1 + 6 **id** 5	\$	r par $F \rightarrow$ **id**
(12) 0 E 1 + 6 F 3	\$	r par $T \rightarrow F$



Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**\$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**\$	d 7
(5) 0 T 2 * 7	**id**+**id**\$	d 5
(6) 0 T 2 * 7 **id** 5	+**id**\$	r par $F \rightarrow$ **id**
(7) 0 T 2 * 7 F 10	+**id**\$	r $T \rightarrow T * F$
(8) 0 T 2	+**id**\$	r $E \rightarrow T$
(9) 0 E 1	+**id**\$	d 6
(10) 0 E 1 + 6	**id**\$	d 5
(11) 0 E 1 + 6 **id** 5	\$	r par $F \rightarrow$ **id**
(12) 0 E 1 + 6 F 3	\$	r par $T \rightarrow F$
(13) 0 E 1 + 6 T 9	\$	r $E \rightarrow E + T$



Pile	Entrée	Action
(1) 0	**id*****id**+**id**	d 5
(2) 0 **id** 5	***id**+**id**\$	r par $F \rightarrow$ **id**
(3) 0 F 3	***id**+**id**\$	r par $T \rightarrow F$
(4) 0 T 2	***id**+**id**\$	d 7
(5) 0 T 2 * 7	**id**+**id**\$	d 5
(6) 0 T 2 * 7 **id** 5	+**id**\$	r par $F \rightarrow$ **id**
(7) 0 T 2 * 7 F 10	+**id**\$	r $T \rightarrow T * F$
(8) 0 T 2	+**id**\$	r $E \rightarrow T$
(9) 0 E 1	+**id**\$	d 6
(10) 0 E 1 + 6	**id**\$	d 5
(11) 0 E 1 + 6 **id** 5	\$	r par $F \rightarrow$ **id**
(12) 0 E 1 + 6 F 3	\$	r par $T \rightarrow F$
(13) 0 E 1 + 6 T 9	\$	r $E \rightarrow E + T$
(14) 0 E 1	\$	accepter

Grammaires SLR

SLR(k) : Simple-LR(k).

- Un item LR(0) (ou simplement item) d'une grammaire G est une production de G avec un point repérant une position dans la partie droite. Une règle $A \to XYZ$ peut donc donner :

$$A \to .XYZ$$

$$A \to X.YZ$$

$$A \to XY.Z$$

$$A \to XYZ.$$

La production $A \to \epsilon$ donne $(A \to .)$. Intuitivement, $A \to XY.Z$ signifie que l'on a déjà lu XY et que l'on s'attend à lire Z pour pouvoir appliquer la production.

Analyse SLR

L'idée de base de l'analyse SLR est de construire un AFD pour reconnaitre les préfixes viables d'une grammaire. Les états de cet AFD correspondent à des ensembles d'items.

- Pour cela on considère une grammaire augmentée. Si G est une grammaire d'axiome S, la grammaire augmentée G' possède l'axiome S' avec $S' \rightarrow S$ et le reste des productions de G.
- La construction d'ensembles d'items équivalents est effectuée par la fonction Fermeture.
- Les transitions entre ces ensembles d'items sont codés par la fonction Transition.

Items valides

Un item $A \rightarrow \beta_1.\beta_2$ est valide pour un préfixe viable $\alpha\beta_1$ si il existe une dérivation telle que :

$$S' \underset{d}{\overset{*}{\Rightarrow}} \alpha A w \underset{d}{\Rightarrow} \alpha\beta_1\beta_2 w$$

Intuitivement un item valide repère une production pouvant nous mener à l'axiome. Il nous informe également de l'action à mener. Si $\beta_2 = \epsilon$, on tend à réduire $A \rightarrow \beta_1$. Inversement si $\beta_2 \neq \epsilon$ on aurait plus tendance à décaler.
Sur la gammaire précédente l'item $T \rightarrow T * .F$ est valide pour le préfixe viable $E + T*$ en effet :

$$E' \underset{d}{\Rightarrow} E \underset{d}{\Rightarrow} E + T \underset{d}{\Rightarrow} E + T * F$$

Dans ce cas là, on aura tendance à décaler pour faire apparaître F et appliquer $T \rightarrow T * F$.

Analyse SLR : Fermeture

Si I est un esemble d'items, $Fermeture(I)$ est défini comme suit :

fonction Fermeture **(**I : ensemble d'items**) :** ensemble d'items
    Déclaration J : Ensemble d'items
début
    J $\leftarrow$ I
    répéter
        Pour chaque $A \rightarrow \alpha.B\beta$ **dans** J **faire**
            Pour chaque $B \rightarrow \gamma$ **dans** G' **faire**
                ajouter $B \rightarrow .\gamma$ à J
            finpour
        finpour
    jusqu'à ce que aucun item n'est ajouté
fin

Intuitivement si l'on s'attend à $\ll$ voir $\gg$ $B\beta$, on doit s'attendre à voir les dérivations possibles $\gamma\beta$ depuis $B\beta$.

Fermeture : Exemple

La grammaire augmentée de l'exemple précédent est :

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E + T | T \\ T & \rightarrow & T * F | F \\ F & \rightarrow & (E) | \mathbf{id} \end{array}$$

et :

$$Fermeture(\{E' \rightarrow .E\}) = \left\{ \begin{array}{lcl} E' & \rightarrow & .E \\ E & \rightarrow & .E + T | .T \\ T & \rightarrow & .T * F | .F \\ F & \rightarrow & .(E) | .\mathbf{id} \end{array} \right.$$

Notion de Noyau

- Notons qu'en dehors de $E' \rightarrow .E$, les items ayant un . à gauche peuvent être retrouvés par application de l'algorithme Fermeture et ne sont donc pas nécessairement stockés.
- Les items devant nécessairement être stockés, i.e. $E' \rightarrow .E$ et tous les items n'ayant pas un . à gauche sont appelés le *Noyau.*

Transition

Transition(I,X) est défini comme suit :

$$J = \{A \to \alpha X.\beta \mid A \to \alpha.X\beta \in I\}$$

et si $J \neq \emptyset$

$$Transition(I, X) = Fermeture(J)$$

Intuitivement si I est l'ensemble des items valides pour un préfixes viable γ, Transition(I,X) est l'ensemble des items valides pour le préfixe viable γX.
Sur l'exemple précédent :

$$Transition(\{T \to T.*F\}, *) = Fermeture(\{T \to T*.F\}) = \begin{cases} T & \to & T*.F \\ F & \to & .(E) \\ F & \to & .\mathbf{id} \end{cases}$$

Construction des ensembles d'items

fonction Items **(**G' : grammaire augmentée**) :** collection d'ensemble d'items
    Déclaration C : collection d'ensemble d'items
début
    C $\leftarrow$ Fermeture($\{S' \rightarrow .S\}$)
    répéter
        Pour chaque I **dans** C **faire**
            Pour chaque symbole X **faire**
                si $Transition(I, X) \neq \emptyset$ et $Transition(I, X) \notin C$ **alors**
                    ajouter Transition(I,X) à C
                finsi
            finpour
        finpour
    jusqu'à ce que aucun ensemble d'items ne soit ajouté à C
fin

Construction des ensembles d'items : Exemple

$$
\begin{array}{lrcl}
I_0 : & E' & \to & .E \\
 & E & \to & .E + T | .T \\
 & T & \to & .T * F | .F \\
 & F & \to & .(E) | .\mathbf{id}
\end{array}
$$

$$
\begin{array}{lrcl}
I_1 : & E' & \to & E. \\
 & E & \to & E. + T
\end{array}
$$

$$
\begin{array}{lrcl}
I_2 : & E & \to & T. \\
 & T & \to & T. * F
\end{array}
$$

$$
\begin{array}{lrcl}
I_3 : & T & \to & F. \\
I_4 : & F & \to & (.E) | .(E) | .\mathbf{id} \\
 & E & \to & .E + T | .T \\
 & T & \to & .T * F | .F
\end{array}
$$

$$
\begin{array}{lrcl}
I_5 : & F & \to & \mathbf{id}.
\end{array}
$$

$$
\begin{array}{lrcl}
I_6 : & E & \to & E + .T \\
 & T & \to & .T * F | .F \\
 & F & \to & .(E) | .\mathbf{id}
\end{array}
$$

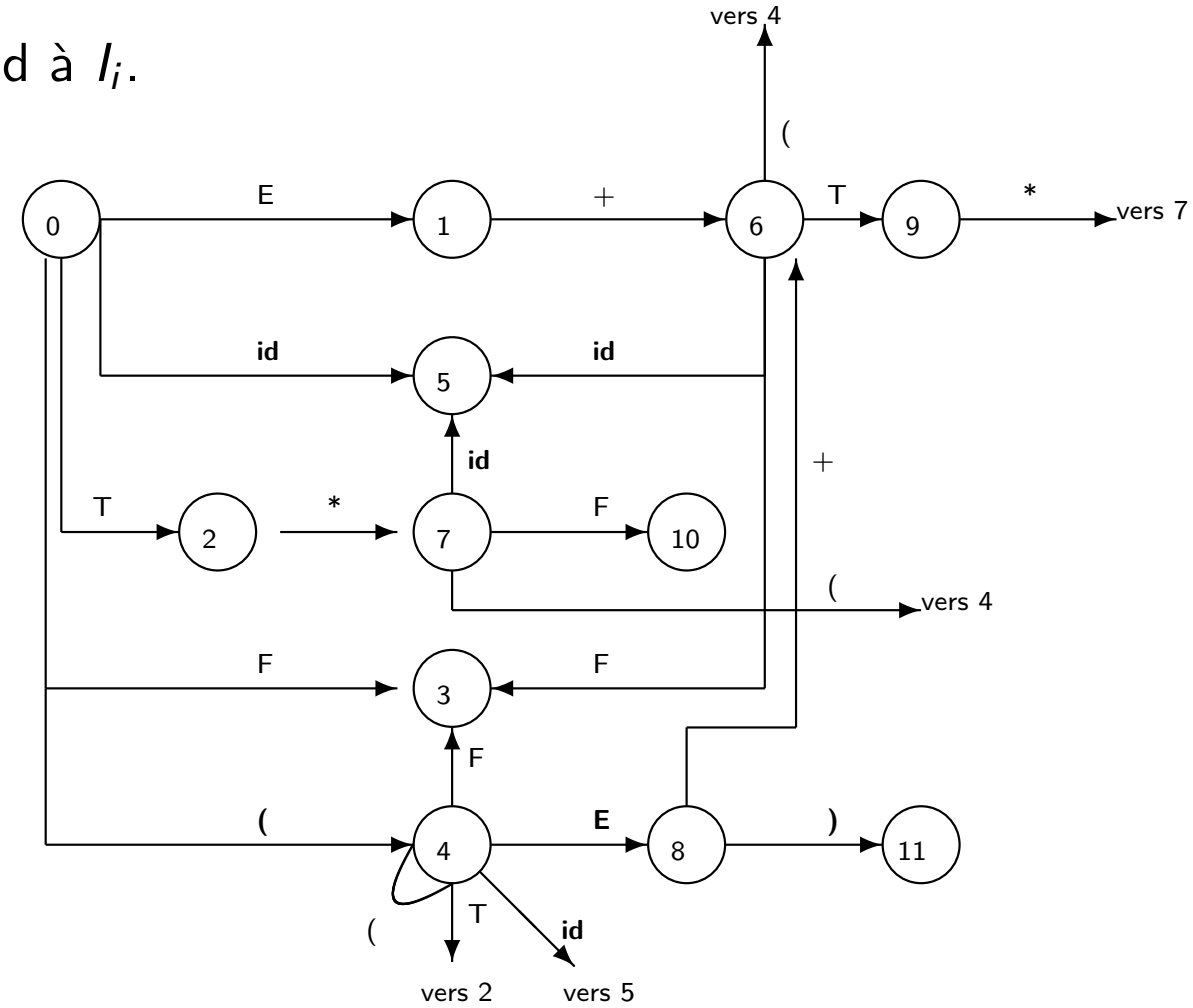
$$
\begin{array}{lrcl}
I_7 : & T & \to & T * .F \\
 & F & \to & .(E) | .\mathbf{id}
\end{array}
$$

$$
\begin{array}{lrcl}
I_8 : & F & \to & (E.) \\
 & E & \to & E. + T
\end{array}
$$

$$
\begin{array}{lrcl}
I_9 : & E & \to & E + T. \\
 & T & \to & T. * F \\
I_{10} : & T & \to & T * F. \\
I_{11} : & F & \to & (E).
\end{array}
$$

Exemple (suite) : l'AFD

L'état i correspond à I_i.



Construction des tables d'analyse SLR

1. C=items(G') (algorithme précédent)
2. Pour chaque état i correspondant à I_i initialiser Action[i,.] comme suit :

2.1 Si $A \to \alpha.a\beta \in I_i$, a terminal remplir Action[i,a] avec décaler j où Transition$(I_i, a) = I_j$.

2.2 Si $A \to \alpha. \in I_i, A \neq S'$, remplir Action[i,a] avec réduire par $A \to \alpha$ pour tout a dans SUIVANT(A).

2.3 Si $S' \to S.$ est dans I_i remplir Action[i,$] par accepter.

Si règles ci-dessus produisent des actions conflictuelles, la grammaire n'est pas SLR(1) : Echec.

3. Pour tout non terminal A si Transition(I_i,A)=I_j alors Successeur[i,A]=j.
4. Les entrées non définies par (2)&(3) correspondent à des erreurs.
5. L'état initial de l'analyseur est celui construit à partir de l'ensemble d'items contenant $S' \to .S$.

Exemple de construction de tables

$$
\begin{array}{lrcl}
I_0: & E' & \to & .E \\
 & E & \to & .E + T | .T \\
 & T & \to & .T * F | .F \\
 & F & \to & .(E) | .\mathbf{id} \\
 & & & \\
I_1: & E' & \to & E. \\
 & E & \to & E. + T \\
 & & & \\
I_2: & E & \to & T. \\
 & T & \to & T. * F \\
I_3: & T & \to & F. \\
I_4: & F & \to & (.E) | .(E) | .\mathbf{id} \\
 & E & \to & .E + T | .T \\
 & T & \to & .T * F | .F \\
I_5: & F & \to & \mathbf{id}.
\end{array}
$$

- Considérons la règle $F \to .(E)$ de I_0. Transiter(I_0,()=I_4. Donc Action[0,(]=d4
- De même, Transiter(I_0,**id**)=I_5 donc Action[0,**id**]=d5.
- Transiter[I_0,E]=I_1 donc Successeur(0,E)=1.
- Transiter[I_0,T]=I_2 donc Successeur(0,T)=2
- Transiter[I_0,F]=I_3 donc Successeur(0,T)=3
- Suivant(E)=$\{\$, +,)\}$, donc Action[2,$]=r par $E \to T$.

On construit petit à petit la table de la page 83.

Limites des grammaires SLR

Considérons la grammaire suivante :

$$\begin{array}{lcl} S & \rightarrow & G = D \\ S & \rightarrow & D \\ G & \rightarrow & *D \\ G & \rightarrow & \textbf{id} \\ D & \rightarrow & G \end{array}$$

qui code l'affectation et l'accès à la valeur d'un pointeur. Un des ensemble d'items de cette grammaire contient :

$$\begin{array}{lcll} I_2 : & S & \rightarrow & G. = D \\ & D & \rightarrow & G. \end{array}$$

Avec $\{=\} \subset \mathit{Suivant}(D)$. Le premier item induit Action[2,=]=décaler alors que le second (puisque $= \in \mathit{SUIVANT}(D)$) induit Action[2,=] réduire par $D \rightarrow G$. La grammaire n'est donc pas SLR(1).

Limite des grammaires SLR : explications

- L'analyseur SLR décide de réduire par $A \to \alpha$ si il rencontre un symbole appartenant à SUIVANT(A). Toutefois, il peut exister des proto-phrase $\beta\alpha$ qui ne peuvent être suivi par un élément de SUIVANT(A). Dans l'exemple précédent, il n'existe pas de proto-phrase $D = D$, la réduction $D \to G$ est donc invalide.
- Il faut donc ajouter plus d'information dans les états. Donné un manche α il nous faut notamment savoir pour quels symboles on peut faire une réduction par A. C'est le but de l'analyse LR.
- Un item LR(1) est un couple $[A \to \alpha.\beta, a]$ avec $a \in \mathit{SUIVANT}(A)$. Sur les items de la forme $[A \to \alpha., a]$ on applique la réduction $A \to \alpha$ uniquement sur lecture du symbole a.
- Notez que l'ensemble des terminaux second membres d'un item $[A \to \alpha.\beta, a]$ sont inclus dans SUIVANT(A), cette inclusion pouvant être stricte (comme dans notre exemple).

Items LR valides

Un item $[A \to \alpha.\beta, a]$ est valide pour un préfixe viable γ si il existe :

$$S \underset{d}{\overset{*}{\Rightarrow}} \delta A w \underset{d}{\Rightarrow} \delta\alpha\beta w$$

tel que :

1. $\gamma = \alpha\beta$
2. a est le premier symbole de w si $w \neq \epsilon$ sinon a=\$.

Fermeture d'items LR(1)

L'analyse LR est essentiellement la même que l'analyse SLR. Seuls les fonctions de fermetures et transition doivent être adaptées.
Donné un item $[A \to \alpha.B\beta, a]$ indiquant que l'on s'attend à voir B (ou ses dérivations) suivi de a qu'elles autres dérivations suivi de quels symboles doit on également s'attendre à voir ? Si l'item est valide pour un préfixe viable γ on a :

$$S \underset{d}{\overset{*}{\Rightarrow}} \delta A a w \underset{d}{\Rightarrow} \delta\alpha B\beta a w \text{ avec } \gamma = \delta\alpha$$

Pour toute réduction $B \to \eta$ on aura $S \underset{d}{\overset{*}{\Rightarrow}} \delta\alpha B\beta a w \underset{d}{\Rightarrow} \delta\alpha\eta\beta a w$.
Par conséquent on doit s'attendre a voir (et donc ajouter à la fermeture) les items de la forme $[B \to .\eta, b]$ avec $b \in PREMIER(\beta a w)$. Puisque a est un terminal, on a $PREMIER(\beta a w) = PREMIER(\beta a)$.

Fermeture d'items LR(1) : Algorithme

fonction Fermeture **(**I : ensemble d'items**) :** ensemble d'items
début
    répéter
        Pour chaque $[A \to \alpha.B\beta, a]$ **dans** I **faire**
            Pour chaque $B \to \eta$ **dans** G' **faire**
                Pour chaque b **dans** $PREMIER(\beta a)$ **faire**
                    si $[B \to .\eta, b] \notin I$ **alors**
                        ajouter $[B \to .\eta, b]$ à I
                    finsi
                finpour
            finpour
        finpour
    jusqu'à ce que aucun nouvel item ne soit ajouté à I
    retourner I
fin

Fermeture d'items LR(1) : exemple (1/2)

Considérons la grammaire augmentée « simple » suivante :

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & CC \\ C & \rightarrow & cC|d \end{array}$$

générant le langage $L = \{c^n d c^m d, n, m \geq 0\}$ et calculons la fermeture de $[S' \rightarrow .S, \$]$. Notons que dans ce cas $\beta = \epsilon$.

- On s'intéresse donc aux items $[S \rightarrow .CC, b]$ avec $b \in PREMIER(\beta\$) = PREMIER(\$) = \{\$\}$. On ajoute donc uniquement l'item $[S \rightarrow .CC, \$]$ (ici $\beta = C$).
- Intéressons nous à présent aux items $[C \rightarrow .cC, b]$ avec $b \in PREMIER(C\$)$. Comme C ne peut se dériver qu'en c ou d, $PREMIER(C\$) = PREMIER(C) = \{c, d\}$. On ajoute donc les items $[C \rightarrow .cC, c]$ et $[C \rightarrow .cC, d]$.
- De même on ajoute $[C \rightarrow .d, c]$ et $[C \rightarrow .d, d]$.

Fermeture d'items LR(1) : exemple (2/2)

- N'ayant plus de non terminaux à droite du point, nous avons fini. Le premier ensemble d'items est donc :

$$
\begin{array}{lll}
I_0 : & S' \to .S, \$ \\
& S \to .CC, \$ \\
& C \to .cC, c/d \\
& C \to .d, c/d
\end{array}
$$

Transition d'items LR(1)

fonction Transition **(**I : ensemble d'items, X non terminal**) :** ensemble d'items
    Déclaration J : Ensemble d'items
début
    J $\leftarrow \{[A \rightarrow \alpha X.\beta, a] \mid [A \rightarrow \alpha.X\beta, a] \in I\}$
    retourner Fermeture(J)
fin

Transition d'items LR(1) : exemple

Calculons les premières transitions de I_0.

- L'item $[S' \to .S, \$]$ n'offre qu'une transition par S pour obtenir$[S' \to S., \$]$. Le point étant à droite on a $\mathit{Fermeture}([S' \to S., \$]) = \{[S' \to S., \$]\}$. On a donc Transition$(I_0, S)$=$I_1$ avec $I_1 = \{[S' \to S., \$]\}$.
- L'item $[S \to .CC, \$]$ nous permet de transiter par C vers la fermeture de $[S \to C.C, \$]$ ce qui nous donne l'état I_2 :

$$\begin{array}{rl} I_2 : & S \to C.C, \$ \\ & C \to .cC, \$ \\ & C \to .d, \$ \end{array}$$

- et ainsi de suite...

Collection d'items LR(1)

fonction Items **(**G' : grammaire augmentée**) :** collection d'ensemble d'items
    Déclaration C : collection d'ensemble d'items
début
    C $\leftarrow$ Fermeture($[S' \rightarrow .S, \$]$)
    répéter
        Pour chaque I **dans** C **faire**
            Pour chaque symbole X **dans** G' **faire**
                si $Transition(I, X) \neq \emptyset$ et $Transition(I, X) \notin C$ **alors**
                    ajouter Transition(I,X) à C
                finsi
            finpour
        finpour
    jusqu'à ce que aucun nouvel ensemble d'item ne soit ajouté à I
    retourner C
fin

Collection d'items : Exemple

La collection d'items pour la grammaire précédente est :

I_0 :
- $S' \to .S$, \$
- $S \to .CC$, \$
- $C \to .cC$, c/d
- $C \to .d$, c/d

I_1 :
- $S' \to S.$, \$

I_2 :
- $S \to C.C$, \$
- $C \to .cC$, \$
- $C \to .d$, \$

I_3 :
- $C \to c.C$, c/d
- $C \to .cC$, c/d
- $C \to .d$, c/d

I_4
- $C \to d.$, c/d

I_5 :
- $S \to CC.$, \$

I_6 :
- $C \to c.C$, \$
- $C \to .cC$, \$
- $C \to .d$, \$

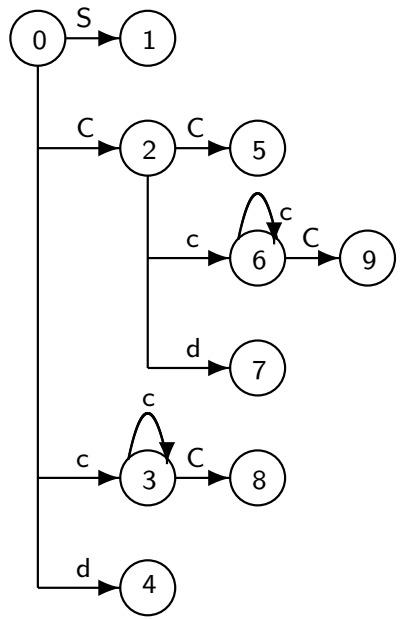
I_7 :
- $C \to d.$, \$

I_8 :
- $C \to cC.$, c/d

I_9 :
- $C \to cC.$, \$

Collection d'items : Exemple

Avec la fonction de transition :



Construction des tables d'analyse LR(1)

1. $C = Items(G')$ (algorithme précédent)
2. Pour chaque état i correspondant à I_i initialiser Action[i,.] comme suit :

2.1 Si $[A \to \alpha.a\beta, b] \in I_i$, a terminal remplir Action[i,a] avec décaler j où Transition$(I_i, a) = I_j$.

2.2 Si $[A \to \alpha., a] \in I_i, A \neq S'$, remplir Action[i,a] avec réduire par $A \to \alpha$.

2.3 Si $[S' \to S., \$]$ est dans I_i remplir Action[i,$] par accepter.

Si les règles ci-dessus produisent des actions conflictuelles, la grammaire n'est pas LR(1) : Echec.

3. Pour tout non terminal A si Transition(I_i,A)=I_j alors Successeur[i,A]=j.
4. Les entrées non définies par (2)&(3) correspondent à des erreurs.
5. L'état initial de l'analyseur est celui construit à partir de l'ensemble d'items contenant $[S' \to .S, \$]$.

Construction des tables d'analyse LR(1) : Remarques

- L'algorithme précédent est très similaire à celui des grammaires SLR(1) page 95.
- La principale différence intervient sur l'item (2.2) où l'on ne se fie plus à suivant puisque l'information sur le prochain caractère attendu est directement stockée dans l'état i.
- Tous les langages de programmation utilisés peuvent être analysés par un analyseur LR(1).
- Par contre, les tables d'analyse LR(1) d'un langage comme C ou Pascal comportent plusieurs miliers d'états alors que les tables SLR(1) correspondantes en ont uniquement quelques centaines.
- On cherche donc une grammaire « presque » aussi puissante que l'analyse LR(1) mais aussi légère que l'analyse SLR(1).
- Il s'agit de l'analyse LALR.

Analyse LALR : Principes

- On constate sur l'automate de la page 108 que de nombreux états ne diffèrent que par leurs terminaux.
- On dit qu'ils ont le même *coeur*.
- L'idée de l'analyse LALR est de fusionner ces états.
- Toutes les liaisons entrantes vers deux états i et j sont donc fusionnées sur l'état ij.
- La fonction de Transition dépendant du coeur et non pas des terminaux si Transition(i,X)=k et Transition(j,X)= l alors k et l devront eux même être fusionnés.

Items fusionnés : Exemple

La collection d'items fusionnés pour la grammaire précédente est :

I_0 :
- $S' \to .S$, \$
- $S \to .CC$, \$
- $C \to .cC$, c/d
- $C \to .d$, c/d

I_1 :
- $S' \to S.$, \$

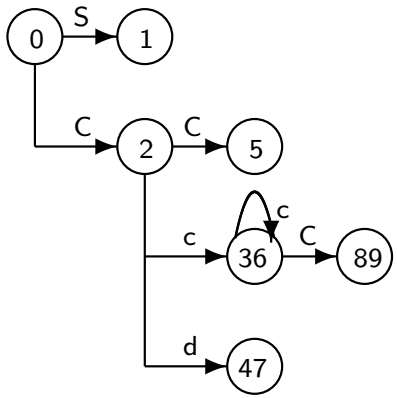
I_2 :
- $S \to C.C$, \$
- $C \to .cC$, \$
- $C \to .d$, \$

I_{36} :
- $C \to c.C$, $c/d/\$$
- $C \to .cC$, $c/d/\$$
- $C \to .d$, $c/d/\$$

I_{47}
- $C \to d.$, $c/d/\$$

I_5 :
- $S \to CC.$, \$

I_{89} :
- $C \to cC.$, $c/d/\$$



Grammaires LALR(1) et conflits décaler/réduire

Aucun conflit décaler/réduire ne peut être induit par la fusion des états de même coeur.

- On a un conflit décaler réduire si dans un même état (fusionné) on a deux productions $[A \to \alpha., a]$ impliquant réduire et $[B \to \beta.a\gamma, b]$ impliquant décaler.
- Soit I un état non fusionné contenant $[A \to \alpha., a]$. Tous les états fusionnés ayant le même coeur, il existe un terminal c tel que $[B \to \beta.a\gamma, c] \in I$. On retrouve donc nos deux items antagonistes qui continuent à produire un conflit décaler/réduire dans un état de l'automate LR(1).
- Cela signifie que la grammaire n'est pas LR(1).

Notons que des conflits réduire/réduire peuvent tout de même apparaître du fait de la fusion.

Construction triviale des tables d'analyse LALR

1. Construire la collection C des items LR(1) pour G'
2. Fusionner les états de C ayant le même coeur.
3. Soit C' le résultat de (2). Si action conflictuelle alors la grammaire n'est pas LALR(1) : Echec.
4. Soit $J = I_1 \cup \cdots \cup I_n$ un ensemble de C' union d'ensemble I_i de C. Si Transition$(I_1, X) = I'_j \neq \emptyset$, soit K tel que $I'_j \subset K \in C'$. Positionner Transition(J,X) à K. Le choix de I_1 est indifférent puisque tous les Transition(I_i, X) sont égaux par construction.
5. Les fonctions Actions et Successeur sont construites à partir de C' comme pour les grammaires LR(1).

   Remarque : cette méthode est loin d'être optimale puisque l'on construit tous les items LR(1) avant de les réduire.

Construction LALR : éviter le calcul de la fermeture

Comme on l'a vu, on peut ne stocker que le noyau I des états (i.e. l'item de l'axiome $[S' \to S, \$]$ et les items où le '.' n'est pas à gauche de la partie droite de la production.

Actions réduire : $A \to \alpha.$ impliquant une action réduire sera dans le noyau à moins que $\alpha = \epsilon$. Réduire $A \to \epsilon$ doit être appelé sur une entrée a si il existe un item $[B \to \gamma.C\delta, b]$ du noyau tel que $C \underset{d}{\overset{*}{\Rightarrow}} A\eta$ avec $a \in PREMIER(\eta\delta b)$. Cet ensemble de non terminaux A peut être pré-calculé.

Actions décaler : Nous décalons sur l'entré a, si il existe $[B \to \gamma.C\delta, b]$ avec $C \underset{d}{\overset{*}{\Rightarrow}} aw$. L'ensemble des terminaux a peut également être pré-calculé.

Transitions : Si $[B \to \gamma.X\delta, b] \in I$ alors $[B \to \gamma X.\delta, b] \in Transition(I, X)$. L'item $[A \to X.\beta, a]$ est également dans $Transition(I, X)$ si $[B \to \gamma.C\delta, b] \in I$ et $C \underset{d}{\overset{*}{\Rightarrow}} A\eta$.

Exemple

Soit la grammaire :

$$\begin{array}{lcl} S & \rightarrow & G = D | D \\ G & \rightarrow & *D | \mathbf{id} \\ D & \rightarrow & G \end{array}$$

Le noyau de sa grammaire LR(0) augmentée est :

$$I_0 : \quad S' \rightarrow .S$$

$$I_1 : \quad S' \rightarrow S.$$

$$I_2 : \quad \begin{array}{lcl} S & \rightarrow & G. = D \\ D & \rightarrow & G. \end{array}$$

$$I_3 : \quad S \rightarrow D.$$

$$I_4 : \quad G \rightarrow *.D$$

$$I_5 : \quad G \rightarrow \mathbf{id}.$$

$$I_6 : \quad S \rightarrow G = .D$$

$$I_7 : \quad G \rightarrow *D.$$

$$I_8 : \quad D \rightarrow G.$$

$$I_9 : \quad S \rightarrow G = D.$$

Grammaire LR(1) correspondante

I_0 : $[S' \to .S; \$]$
$[S \to .G = D; \$]$
$[G \to . * D; =, \$]$
$[G \to .id; =, \$]$
$[S \to .D; \$]$
$[D \to .G; \$]$

I_1 : $[S' \to S.; \$]$

I_2 : $[S \to G. = D; \$]$
$[D \to G.; \$]$

I_3 : $[S \to D.; \$]$

I_4 : $[G \to *.D; =, \$]$
$[D \to .G; =, \$]$
$[G \to . * D; =, \$]$
$[G \to .id; =, \$]$

I_5 : $[G \to \mathbf{id}.; =, \$]$

I_6 : $[S \to G = .D; \$]$
$[D \to .G; \$]$
$[G \to . * D; \$]$
$[G \to .id; \$]$

I_7 : $[G \to *D.; =, \$]$

I_8 : $[D \to G.; =, \$]$

I_9 : $[S \to G = D.; \$]$

Génération et propagation des symboles de pré vision

Si l'item LR(0) $B \to \gamma.C\delta$ est dans le noyau de I, on rappelle que $A \to X.\beta \in \mathit{Transition}(I, X)$ si $C \underset{d}{\overset{*}{\Rightarrow}} A\eta$

- Supposons à présent que l'item LR(1) $[B \to \gamma.C\delta, b] \in I$.Puisque $C \underset{d}{\overset{*}{\Rightarrow}} A\eta$, l'item $[A \to X.\beta, a] \in \mathit{Transition}(I, X)$ pour tout $a \in \mathit{PREMIER}(\eta\delta)$. On dit que a est engendré spontanément. Le terminal \$ est engendré spontanément pour $[S' \to .S]$.
- Toutefois si $\eta\delta \underset{d}{\overset{*}{\Rightarrow}} \epsilon$, alors $[A \to X.\beta, b] \in \mathit{Transition}(I, X)$. On dit dans ce cas, que le symbole b se propage.
- l'algorithme suivant calcule les symboles de prévision en utilisant le symbole fictif $\#$ pour détecter les dérivations $\eta\delta \underset{d}{\overset{*}{\Rightarrow}} \epsilon$.

Détermination des symboles de pré vision : Algorithme

procédure SymbolesPreVision **(**E I : un état,X un symbole, S PreVision**)**
début
J $\leftarrow$ Transition(I,X)
K $\leftarrow$ ensemble d'items LR(0) de I
Pour chaque $B \rightarrow \gamma.\delta$ **dans** K **faire**
J' $\leftarrow$ Fermeture($[B \rightarrow \gamma.\delta, \#]$)
si $[A \rightarrow \alpha.X\beta, a] \in J'$ et $a \neq \#$ **alors**
Remarque : *Génération de a*
PreVision[J,$A \rightarrow \alpha X.\beta$] $\leftarrow$ PreVision[J,$A \rightarrow \alpha X.\beta$] $\cup \{a\}$
finsi
si $[A \rightarrow \alpha.X\beta, \#] \in J'$ **alors**
Remarque : *Propagation des b*
PreVision[J,$A \rightarrow \alpha X.\beta$] $\leftarrow$
PreVision$[J, A \rightarrow \alpha X.\beta] \cup \{b \mid [B \rightarrow \gamma.\delta, b] \in I\}$
finsi
finpour
fin

Symboles de prévision

Reprenons la grammaire présentée page 116 et calculons Fermeture($[S' \rightarrow .S, \#]$). On obtient :

$$\begin{array}{lcl} S' & \rightarrow & .S, \# \\ S & \rightarrow & .G = D, \# \\ S & \rightarrow & .D, \# \\ G & \rightarrow & . * D, \# / = \\ G & \rightarrow & .\mathbf{id}, \# / = \\ D & \rightarrow & .G, \# \end{array}$$

L'item $[G \rightarrow . * D, =]$ implique une propagation du $=$ vers $G \rightarrow *.D$ de I_4. De même, $[G \rightarrow .\mathbf{id}, =]$ propage le $=$ vers $G \rightarrow \mathbf{id}.$ de I_5. Le symbole $\#$ est propagé à I_1 sur lecture de S, I_2 sur lecture de G, I_3 sur lecture de D et ajouté à I_4 et I_5 sur lecture de respectivement * et **id**.

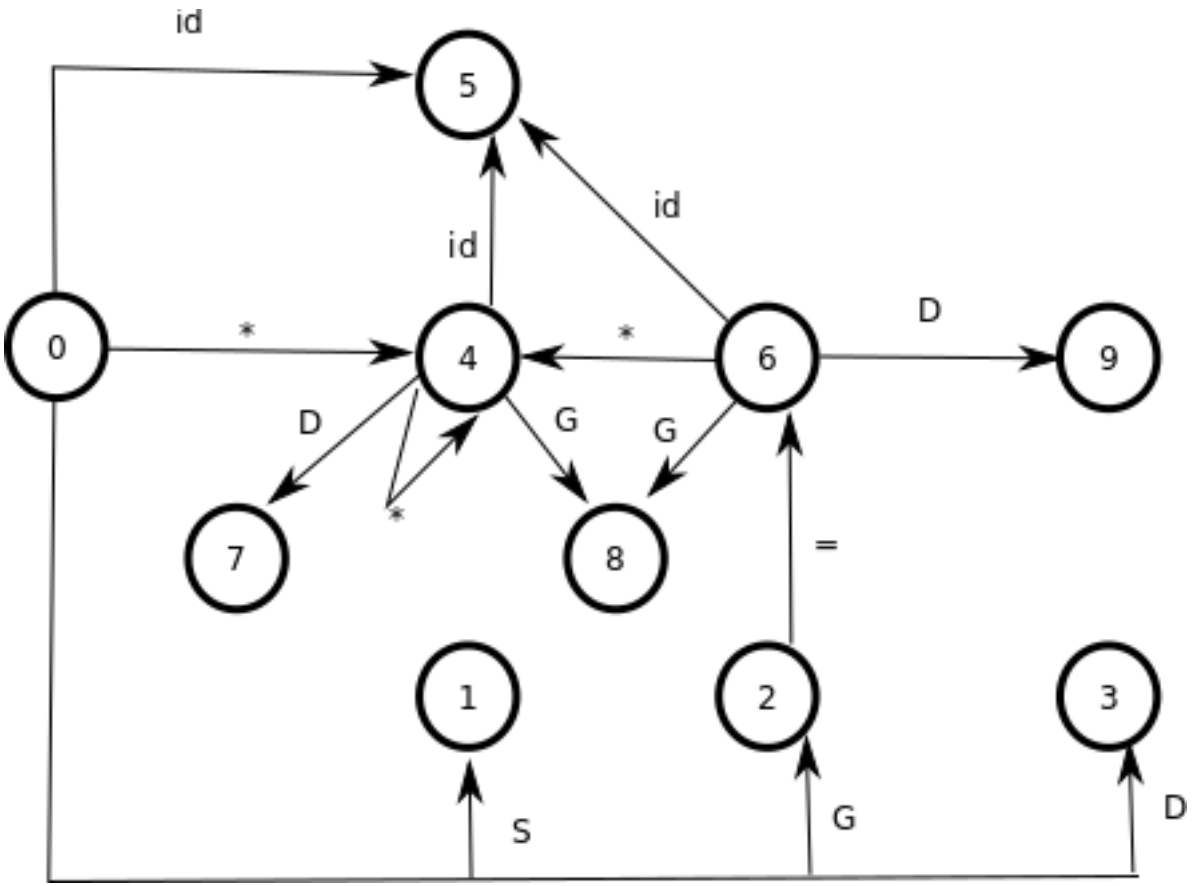
Propagation des symboles de pré vision

procédure CalculNoyaux **(**E G : grammaire,S C : collection d'ensemble d'items LR(0), PreVision : table de symboles de prévision**)**
début
    C ← Collection d'items LR(0) de G
    répéter
        Pour chaque I **dans** C **faire**
            Pour chaque symbole X **dans** G **faire**
                SymbolesPrevision(I,X,PreVision)
            finpour
        finpour
    jusqu'à ce que aucune modification de la table PreVision
fin

Transmission des pré vision

De	*Vers*
$I_0 : \quad S' \to .S$	$I_1 : \quad S' \to S.$
	$I_2 : \quad S \to G. = D$
	$\quad\quad D \to G.$
	$I_3 : \quad S \to D.$
	$I_4 : \quad G \to *.D$
	$I_5 : \quad G \to \mathbf{id}.$
$I_2 : \quad S \to G. = D$	$I_6 : \quad S \to G = .D$
$I_4 : \quad G \to *.D$	$I_4 : \quad G \to *.D$
	$I_5 : \quad G \to \mathbf{id}.$
	$I_7 : \quad G \to *D.$
	$I_8 : \quad D \to G.$
$I_6 : \quad S \to G = .D$	$I_4 : \quad G \to *.D$
	$I_5 : \quad G \to \mathbf{id}.$
	$I_8 : \quad D \to G.$
	$I_9 : \quad S \to G = D.$

Transmission des pré vision : Le diagramme



Transmission des pré vision : Les différentes passes

Ensemble	*Item*	*Previsions*			
		Init	*Passe*1	*Passe*2	*Passe*3
I_0 :	$S' \to .S$	\$	\$	\$	\$
I_1 :	$S' \to S.$		\$	\$	\$
I_2 :	$S \to G. = D$		\$	\$	\$
	$D \to G.$		\$	\$	\$
I_3 :	$S \to D.$		\$	\$	\$
I_4 :	$G \to *.D$	$=$	$=/\$$	$=/\$$	$=/\$$
I_5 :	$G \to \mathbf{id}.$	$=$	$=/\$$	$=/\$$	$=/\$$
I_6 :	$G \to G = .D$			\$	\$
I_7 :	$G \to *D.$		$=$	$=/\$$	$=/\$$
I_8 :	$D \to G.$		$=$	$=/\$$	$=/\$$
I_9 :	$S \to G = D.$				\$

Un outil d'analyse grammaticale : Yacc

- Lex retourne des tokens qui sont combinés dans Yacc par une grammaire pour reconnaitre un langage.
- Les cousins de Lex/Yacc pour le C++ se nomment Flex et Bisons.
- L'application de Yacc sur un fichier produit un fichier `y.tab.c`.
- Ci joint un exemple de Makefile :

```
CC      = gcc
LEX     = lex
YACC    = yacc
LEXFILE = monFichier.lex
YACCFILE = monFichier.y
OBJ     = y.tab.o
OUTPUT  = monFichier
LIB     = -ll

$(OUTPUT) : $(OBJ)
	$(CC) $(OBJ) -o $@ $(LIB)

y.tab.o : lex.yy.c y.tab.c
	$(CC) -c y.tab.c

lex.yy.c : $(LEXFILE)
	$(LEX) $(LEXFILE)

y.tab.c : $(YACCFILE)
	$(YACC) $(YACCFILE)
```

Combinaison de Lex et Yacc

- Pour un projet un peu important on peut générer le .o de lex.yy.c puis le .o de y.tab.c et linker les deux (ce n'est pas ce qui est fait dans le makefile précédent).
- Pour de petits compilateurs/interpréteurs (éventuellement dans de grands projets), il est d'usage d'inclure le fichier lex.yy.c dans le fichier Yacc et ainsi de compiler les deux simultanément (voir plus loin).

Structure d'un fichier Yacc

%{
Déclaration des variables C
%}
Déclaration de tokens et d'associativité/priorité des opérateurs
%%
Déclaration des productions et de leurs actions associées
%%
Code C supplémentaire.
Par exemple :

```
#include "lex.yy.c"

main()
{
  return yyparse();
}

int yyerror( char *s ) { fprintf( stderr, "Error : %s\n", s); }
```

Yacc : un exemple (fichier Lex)

```
%{
#include <stdio.h>
%}
blanc [ ]
pm [+-]
chiffre [0-9]
frac \.{chiffre}+
exp E{pm}?{chiffre}+
nb {pm}?({chiffre}+{frac}?|{frac}){exp}?
%%
{blanc} {}
{nb} {yylval=atof(yytext); return NB;}
- {return NEG;}
\n|. {return yytext[0];}
%%
```

Yacc : un exemple (fichier Yacc (1/3))

Considérons la grammaire des expressions arithmétiques suivante :

$$\begin{array}{lcl} E & \rightarrow & E + T | E - T | T \\ T & \rightarrow & T * F | T / F | F \\ F & \rightarrow & (E) | - E | \textbf{nombre} \end{array}$$

Yacc : un exemple (fichier Yacc (2/3))

```
%{
  #include <stdio.h>
  #define YYSTYPE double
%}
%token NB
%token NEG
%%
input:
/* ligne vide */
|input ligne
;
ligne : expr '\n' {printf("%f\n",$1);}
;
expr : expr '+' terme {$$=$1+$3;}
| expr NEG terme {$$=$1-$3;}
| terme
;
```

Yacc : un exemple (fichier Yacc (3/3))

```
terme : terme '*' facteur {$$=$1*$3;}
| terme '/' facteur {$$=$1/$3;}
| facteur
;
facteur: '('expr')' {$$=$2;}
| NEG expr   {$$=-$2;}
| NB
;
%%
#include "lex.yy.c"

main()
{
  return yyparse();
}

int yyerror( char *s ) { fprintf( stderr, "Error : %s\n", s); }
```

Exemple Yacc : Explications

- La ligne #define YYSTYPE double, permet de spécifier le type des valeurs que l'on va affecter aux symboles de notre grammaire.
- La ligne input, spécifie qu'une input est soit une ligne vide, soit un input suivi d'une ligne. Ceci permet de boucler sur la lecture de l'entrée. Notez la structure d'une production :

1. Le membre gauche est avant le :
2. les différentes productions concernant ce membre gauche sont séparées par des |.
3. La production se termine par un ;

- La production 'ligne :' est dérivée sur lecture d'un retour chariot et affiche le résultat de l'évaluation de la ligne. Notez l'utilisation du $1 qui représente le premier symbole du membre droit ($2 représentera le second, $3 le troisième et ainsi de suite).

Exemple Yacc : Explications

- La première production expr est appellée sur dérivation d'une addition. Dans ce cas on affecte à la valeur du membre gauche ($$) les valeurs $1 et $3 représentant respectivement les valeurs de expr et terme.
- La fonction main() appelle tout simplement le parseur. Il ne faut bien sur pas définir de main dans le fichier Lex.

Yacc et les grammaires ambiguës

Yacc peut aussi considérer des grammaires ambiguë à condition que l'on lève les ambiguïtés via la définition des règles d'associativité et de priorité des opérateurs.

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token NOMBRE
%token PLUS MOINS FOIS DIVISE PUISSANCE
%token PARENTHESE_GAUCHE PARENTHESE_DROITE
%token FIN

%left PLUS MOINS
%left FOIS DIVISE
%left NEG
%right PUISSANCE

% start Input
%%
Input:
    /* Vide */
  | Input Ligne
  ;
Ligne:
    FIN
  | Expression FIN    { printf("Resultat : %f\n",$1); }
  ;
```

```
Expression:
    NOMBRE      { $$=$1; }
  | Expression PLUS Expression  { $$=$1+$3; }
  | Expression MOINS Expression { $$=$1-$3; }
  | Expression FOIS Expression  { $$=$1*$3; }
  | Expression DIVISE Expression  { $$=$1/$3; }
  | MOINS Expression %prec NEG  { $$=-$2; }
  | Expression PUISSANCE Expression { $$=pow($1,$3); }
  | PARENTHESE_GAUCHE Expression PARENTHESE_DROITE  { $$=$2; }
  ;
%%
```

Grammaires ambiguë : Explications

- `%left PLUS MOINS` indique que + et - sont associatifs à gauche et d'égale priorité
- `%left FOIS DIVISE` indique que * et / sont associatifs à gauche, d'égale priorité et de priorité supérieure à + et -.
- `%left NEG` sert surtout à indiquer que l'opérateur moins unaire est de priorité supérieure à * et /. Notez que la token `NEG` n'existe pas. On indique simplement une priorité.
- Enfin, la puissance est associative à droite et de priorité la plus importante.
- la présence de `%prec NEG` dans le membre droit de la production `Expression`, indique que cette production a la priorité de `NEG` (donc entre *,/ et puissance).