ESIAL – TELECOM NANCY 2012 - 2013

Cours de Traduction I

Deuxième année

Partie n°1 : chapitres 1, 2 et 3.

Table des matières

Chapitre n°1 : introduction 2

Qu'est-ce qu'un compilateur ? 2

Structure d'un compilateur 3

Chapitre n°2 : Analyse lexicale 4

Définition et fonctionnement 4

Codage des tokens et lexique 5

Chapitre n°3 : Analyse syntaxique 6

Analyse descendante 6

Analyse ascendante 8

1) Fonctionnement 8

2) Définitions 10

3) Analyseur LR(0) 11

4) Analyseur SLR(1) 12

5) Analyseur LR(1) 15

6) Analyse LALR(1) 17

7) Analyse ascendante et grammaire ambigüe 17

8) Fonctions sémantiques en analyse syntaxique ascendante 20

Conclusion 22

Chapitre n°1 : introduction

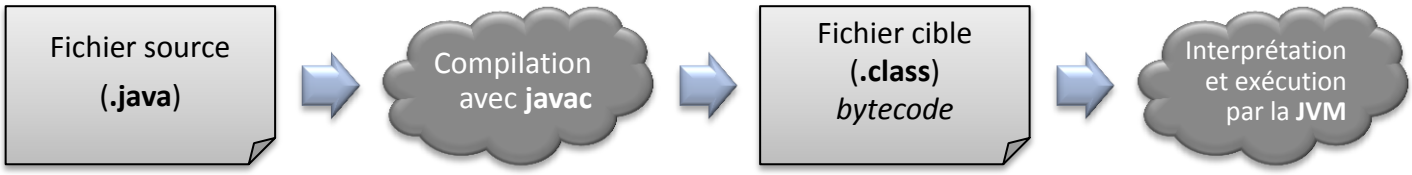
Qu'est-ce qu'un compilateur ?

Un compilateur est un logiciel qui traduit un programme écrit dans un langage de haut niveau, le code source, dans un autre langage (de haut niveau) ou bien en instructions exécutables par une machine donnée (en langage machine, de bas niveau). Il doit pouvoir signaler les éventuelles erreurs contenues dans le code source.

Différence entre un compilateur et un interpréteur

- Le **compilateur** lit le code source fourni ; le fichier résultat, contenant le code machine (appelé aussi fichier cible), est exécutable une fois pour toute.
  Le C, le C++, le Fortran, le Pascal, le Cobol, l'Ada… sont des langages compilés.
- L'**interpréteur** ne crée pas de fichier cible. Il interprète et exécute au fur et à mesure les instructions du programme source.
  L'OCaml, le PHP, le Python, le sh, le bash… sont des langages interprétés.

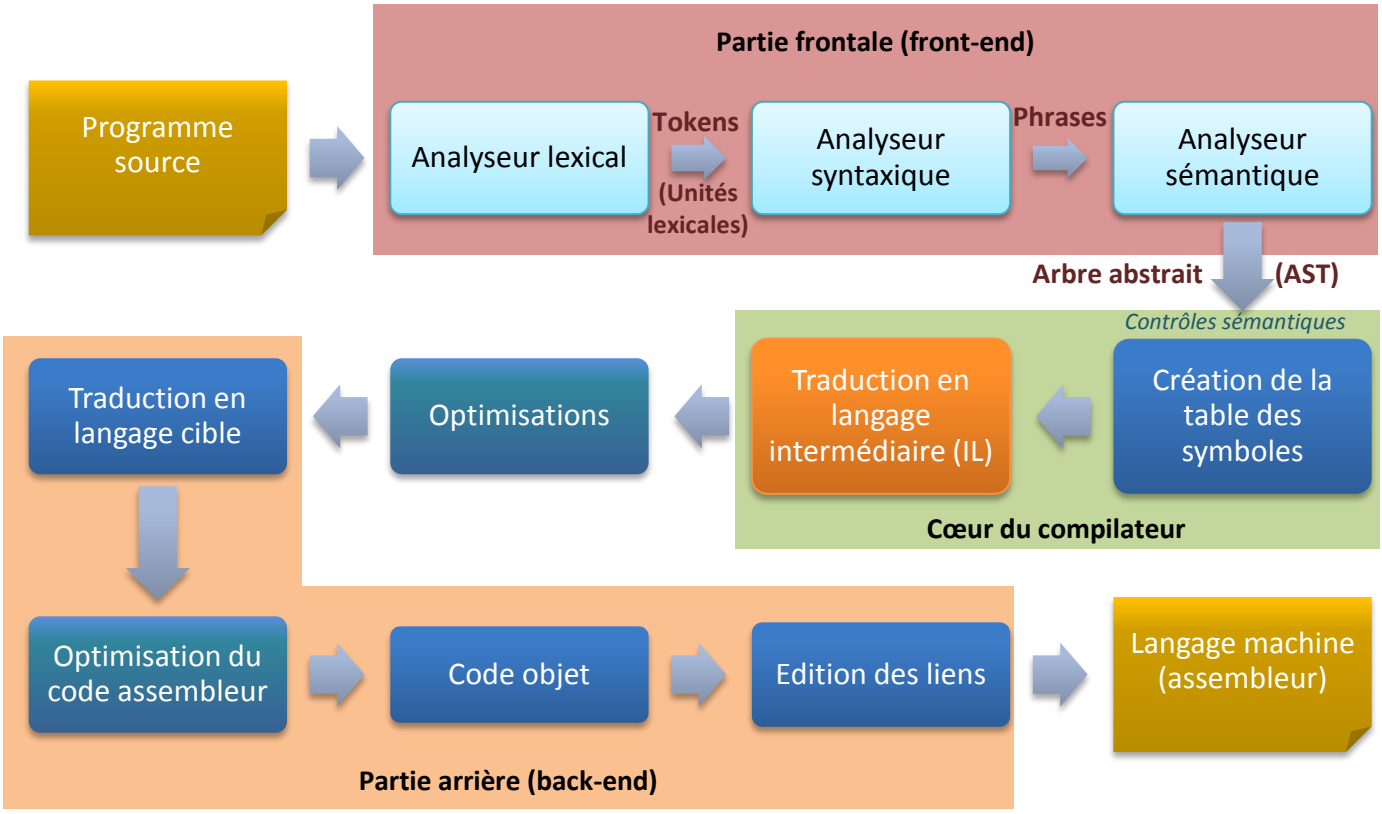
Certains langages ne sont ni compilés, ni interprétés totalement (c'est un « mélange » des deux) : c'est le cas du langage **Java**. En effet, le résultat de la compilation d'un code source java est un fichier non pas en langage machine, mais en un *bytecode java* qui n'est pas directement exécutable par la machine. Le *bytecode* généré par le compilateur *javac* est interprété et exécuté par la machine virtuelle java : la **JVM**.



Quelques dates

- 1954 : Fortran (le premier compilateur date des années 1950)
- 1958 : Lisp (langage fonctionnel)
- 1959 : Cobol (langage « objet »)
- 1966 : Stimula (classes et objets)
- 1971 : Pascal
- 1972 : C, Smalltalk, Prolog
- 1976 : Ada
- 1982 : C++
- 1986 : Eiffel
- 1991 : Java (par Sun)

Structure d'un compilateur

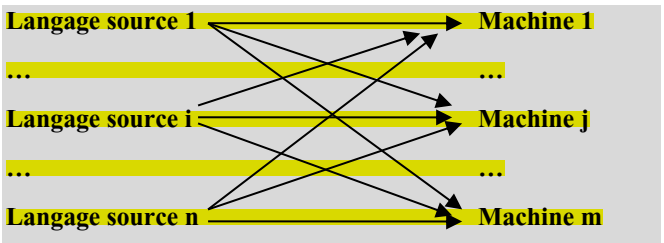


Intérêt du langage intermédiaire

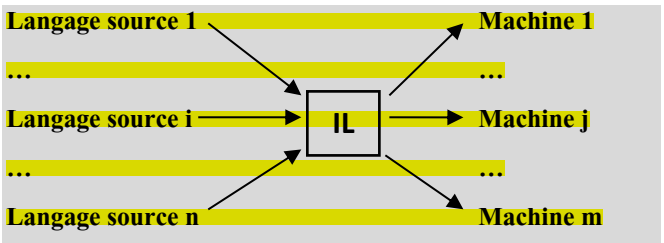
De nombreux compilateurs, pour traduire le langage source dans le langage cible, utilisent un langage intermédiaire. Ce langage intermédiaire est théoriquement indépendant à la fois du langage source et du langage cible.

En effet, lorsqu'on décide d'écrire un compilateur pour un langage source donné, il faudra pouvoir générer le langage assembleur de chaque machine (une machine ayant un jeu d'instructions assembleur spécifique). Utiliser un langage intermédiaire évite de devoir réécrire toute la partie frontale du compilateur à chaque nouvelle machine. Seule la partie arrière sera adaptée pour pouvoir générer le bon langage assembleur.

De même, si l'on souhaite faire un compilateur pour d'autres langages source, seule la partie frontale sera à réécrire, puisque le langage intermédiaire sera le même, et ainsi la partie arrière effectuera le même traitement pour une même machine.



*Sans langage intermédiaire : **n×m** compilateurs à écrire*



*Avec langage intermédiaire : **n+m** compilateurs à écrire*

Chapitre n°2 : Analyse lexicale

Définition et fonctionnement

- C'est l'unique module du compilateur qui lit le fichier contenant le code source. Il le lit caractère par caractère.
- Il reconnaît les unités lexicales (ou tokens) qui sont les mots du langage et les présente à l'analyseur syntaxique.
- C'est un **automate**.

Exemple :

```
int x;
if (x>10)
        then print(x) ;
```

Tokens: "**int**" "**x**" "**;**" "**if**" "**(**" "**x**" "**>**" "**10**" "**)**" "**then**" …

Les principaux types d'unités lexicales reconnues par l'analyseur sont les suivants :

- caractères spéciaux simples : "**+**" "**-**" "*****" "**/**" "**<**" "**>**" "**%**" "**;**" "**(**" "**)**" …
- caractères spéciaux doubles : "**++**" "**--**" "**<=**" "**>=**" " **!=** " …
- mots-clés: "**if**" "**for**" "**then**" …
- constantes : "**123**" "**-0,5**" …
- identificateurs (*idf*): "**x**" "**i**" "**fact**" "**printf**" "**main**"

Exemple :

```
x := y + 2 * x
```

(idf)	(car. spé.)	(idf)	(car. spé.)	(constante)	(car.spé.)	(idf)
x	**:=**	**y**	**+**	**2**	*****	**x**

Pour reconnaître ces tokens, on utilise les expressions régulières (comme en *shell* avec **grep** et **sed**).

Les autres tâches de l'analyseur sont les suivantes :

- Eliminer les commentaires
- Enlever les espaces, tabulations, retour à la ligne
- Afficher les erreurs lexicales (caractère inconnu, identificateur trop long, constante mal formée…)
- Découper le programme source en unités lexicales, qui seront passées l'une après l'autre à l'analyseur syntaxique.

Codage des tokens et lexique

L'analyseur lexical code les unités lexicales reconnues pour des raisons d'efficacité. En effet, avec le lexique, il est plus facile de manipuler des codes entiers (*int*) que des chaînes de caractères.

Exemple simple :

- Unités lexicales simples :
  - "**+**" *est codé par* **1**
  - "**-**" *est codé par* **2**
  - "*****" *est codé par* **3**
  - "**/**" *est codé par* **4**
  - "**:=**" *est codé par* **5**
- Unités lexicales génériques :
  - **idf** *(identifiant) est codé par* **7**
  - **cste** *(constante) est codé par* **8**
- Mots-clés du langage :
  - "**if**" *est codé par* **9**
  - "**then**" *est codé par* **10**

Remarque : *en ce qui concerne les unités lexicales simples, on aurait pu se servir du code ascii : à chaque caractère correspondant à une unité lexicale simple on aurait associé son code ascii.*

Programme source:
x := y + 2 * x

En utilisant les codes ci-dessus, l'analyseur lexical renverra alors :

(7, 'x') 5 (7, 'y') 1 (8, 2) 3 (7, 'x')

Le lexique

Or, pour éviter de répéter plusieurs fois la même chaîne de caractères dans la sortie de l'analyseur lexical, on se sert du lexique pour associer un entier à chacune des chaînes de caractères.
Ces entiers associés aux chaînes de caractères sont bien sûr indépendants de ceux associés aux *tokens*.

Lexique (table)	
Chaîne de caractère	**Index associé (entier)**
"x"	1
"y"	2

Il y a une entrée dans le lexique par chaîne de caractère. L'index s'appelle le numéro lexical.

Programme source:
x := y + 2 * x

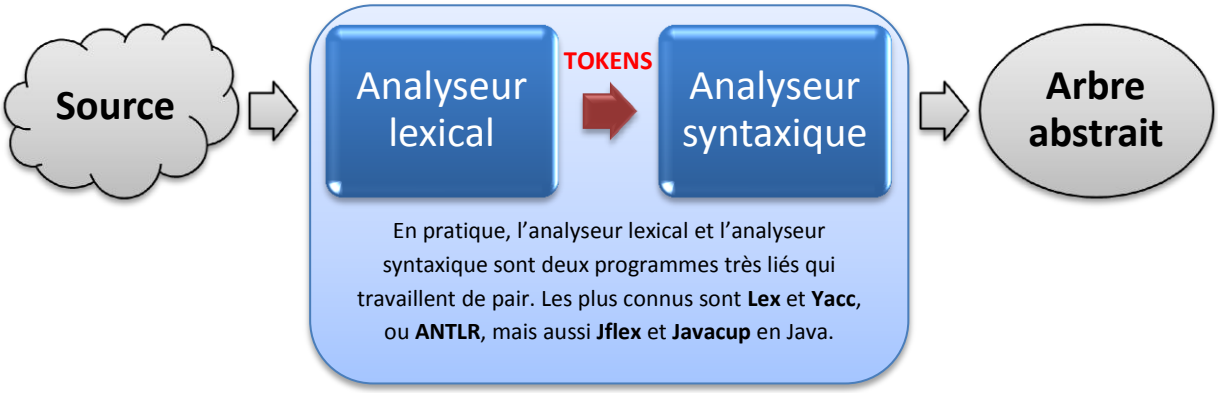
Par conséquent, la sortie de l'analyseur devient :

(7, 1) 5 (7, 2) 1 (8, 2) 3 (7, 1)

Chapitre n°3 : Analyse syntaxique

But de l'analyse syntaxique

- ✓ Vérifier que la phrase en entrée est bien formée, c'est-à-dire vérifie la syntaxe du langage définie par une grammaire
- ✓ Reconnaître les unités syntaxiques décrites par une grammaire (déclarations, affectations, instructions, variables, expressions, fonctions, procédures…)
- ✓ Détecter les erreurs syntaxiques : afficher des messages clairs et précis contenant le numéro de ligne de l'erreur, corriger l'erreur si possible, et poursuivre l'analyse syntaxique pour détecter d'autres erreurs
- ✓ L'analyse d'un programme correct ne doit pas être ralentie.



Il existe deux types d'analyseurs syntaxiques : **l'analyseur syntaxique descendant** (ASD) tel que ANTLR, ou bien **l'analyseur syntaxique ascendant** (ASA) tel que Lex-Yacc.

Analyse descendante

- ❖ Le code source est lu unité lexicale par unité lexicale
- ❖ L'arbre syntaxique est construit à partir de l'axiome, en accrochant les parties droites des productions sous la racine (suite d'expansions).

Rappels

- Dans la définition d'une grammaire, les non-terminaux sont en majuscules, les terminaux en minuscules.
- Pour effectuer une analyse syntaxique descendante, la grammaire doit être **LL(k).** Le premier '**L**' signifie ***L****eft to Right Scanning* ; le second signifie ***L****eft Most Derivation*. Ainsi, on utilise les **k** prochaines unités lexicales pour décider de la production à appliquer lors de l'expansion.
- Une grammaire **LL(k)** n'est jamais ambiguë.
- Une grammaire récursive gauche n'est **LL(k)** pour aucun **k**.
- Le mot vide, noté **^** ou **ε**, n'est pas une unité lexicale.

Il y a deux catégories d'analyseurs syntaxiques descendants :

- par fonctions récursives (*cf*. TD)
- par table d'analyse syntaxique (comme vu en cours de MI2 de première année, utilisé par ANTLR)

Analyse syntaxique descendante par fonctions récursives :

- A chaque non-terminal on associe une fonction qui vérifie que le texte correspond à la syntaxe décrite par les différentes alternatives
- Pour choisir entre ces différentes alternatives, les fonctions disposent d'une unité lexicale d'avance.

Exemple :

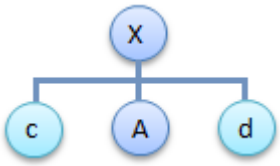
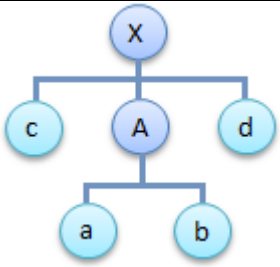
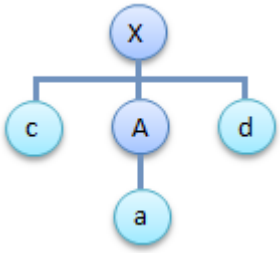
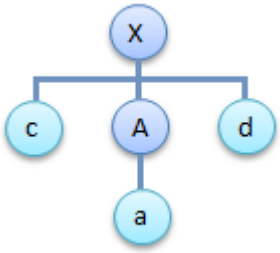
Grammaire:

X → cAd

A → ab | a

Texte source:

cad$

Fonctionnement d'un ASD par fonctions récursives		
Etape n°	*Arbre*	*Texte lu*
1		**cad$**
2		**ad$**
3	**ERREUR :** **retour en arrière**	**d$**
4		**d$**
5	 **SUCCES**	**$**

Analyse ascendante

L'analyse syntaxique ascendante ne fonctionne que par table d'analyse déterministe.

- ✓ On part du noyau à analyser
- ✓ On remplace itérativement des fragments du mot courant qui correspondent à des parties droites de règles de production par le membre gauche de cette règle.

Les analyseurs ascendants sont prédictifs : **LR(k)**. Le '**L**' correspond à ***L**eft to Right Scanning*, le '**R**' à ***R**ight Most Derivation*. On utilise **k** symboles du mot à analyser pour faire la prédiction. Trois types d'ASD seront étudiés dans ce cours :

- **LR(0)**
- **SLR(1)**
- **LR(1)** / **LALR(1)** (comme l'analyseur syntaxique ***Yacc***)

Ces analyseurs sont basés sur **deux actions** :

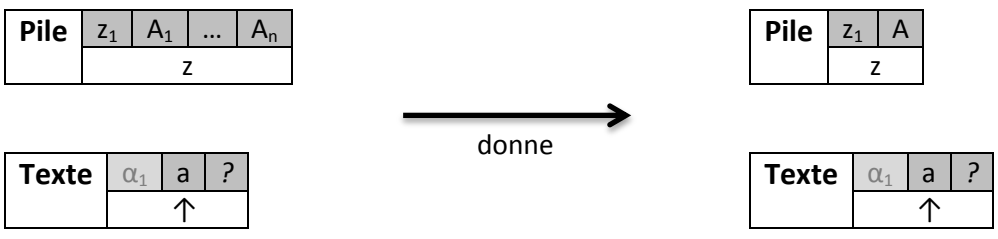
- ❖ **La réduction (REDUCE)** : remplacer un motif par son non-terminal.
- ❖ **Le décalage (SHIFT)** : lecture dans le texte.

1) Fonctionnement

- ✓ Construire une dérivation en remontant vers l'axiome
- ✓ Construire l'arbre de dérivation :
  - racine de l'arbre : axiome
  - nœud interne : non-terminal
  - feuille : terminal
  - passage nœud interne / "successeur" : règle
- ✓ On dispose :
  - d'une pile qui contient une phrase $z \in \{N \cup T\}^*$ (vide au départ)
  - la phrase à analyser **α$** telle que **α$ = α_1 a … $**, avec **a** l'unité lexicale courante, et **α_1** la partie déjà analysée et dérivée de ***z***.

Principe n°1 : REDUCTION (*ou REDUCE*)

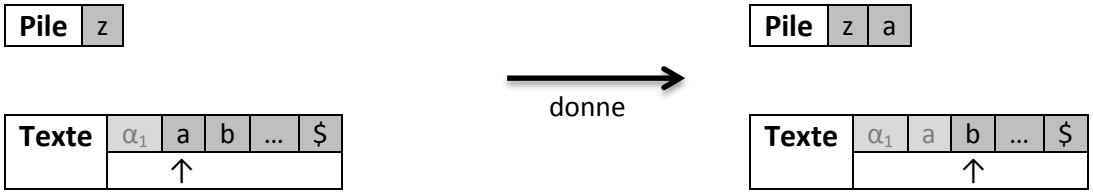
On reconnaît que la pile se termine par un membre gauche **A_1 … A_n** d'une règle **A → A_1 … A_n**. Alors on remplace cette fin de pile par A.



On n'avance pas dans le texte source.

Principe n°2 : LECTURE (*ou SHIFT*)

On avance dans le texte source, c'est-à-dire on lit, et on ajoute a (l'unité lexicale courante) dans la pile, et on lit l'unité lexicale suivante.



Si le texte se réduit au marqueur de fin (**$**) et si la pile se réduit à l'axiome, alors le texte était correct (**succès**).

Exemple d'analyse syntaxique ascendante :

Grammaire:

S → Ac
A → AaAb | d

On augmente la grammaire :
(*Axiome' → Axiome*)

Grammaire augmentée:

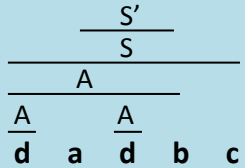
S' → S
S → Ac
A → AaAb | d

Texte source:
dadbc$

Fonctionnement général d'un ASA			
Etape n°	*Action effectuée*	*Texte source*	*Pile*
0	-	**dadbc$**	☐ *(vide)*
1	LECTURE	**dadbc$**	d ☐
2	REDUCTION **par A→d**	**dadbc$**	A ☐
3	LECTURE	**dadbc$**	A a ☐
4	LECTURE	**dadbc$**	A a d ☐
5	REDUCTION **par A→d**	**dadbc$**	A a A ☐
6	LECTURE	**dadbc$**	A a A b ☐
7	REDUCTION **par A→AaAb**	**dadbc$**	A ☐
8	LECTURE	**dadbc$**	A c ☐
9	REDUCTION **par S→Ac**	**dadbc$**	S ☐
10	REDUCTION **par S'→S**	**dadbc$**	S' ☐

SUCCES
car la pile se réduit à l'axiome et le mot vaut **$**

Arbre syntaxique



Problématique :

Comment décider par algorithme de l'action à effectuer à chaque étape, à savoir ***REDUCE***, ou ***SHIFT*** ?

Pour cela, on construit un automate déterministe **LR(0)** qui dira ce qu'il faut faire en fonction du contenu de la pile, et de l'unité lexicale courante.

2) Définitions

Item

Un item d'une grammaire G est une production de G avec un marqueur (noté •) repérant une position dans la partie droite.

Par exemple : pour la règle **S → Ac** : on peut avoir **S → A•c**
ou **S → •Ac**
ou **S → Ac•**

Remarque : si **A → ε** on ne note pas **A → •ε** ou **A → ε•** , car le mot vide n'est pas un terminal. On note **A → •**.

Fermeture

Soit **I** un ensemble d'items de G.

La fermeture de **I**, notée **Fermeture(I)**, est un ensemble d'items construits à partir de **I** tel que :

- ✓ placer chaque item de I dans Fermeture(I)
- ✓ si **A → α•Bβ** ∈ **Fermeture(I)** et **B → γ**, alors ajouter **B → •γ** à **Fermeture(I)** (sauf si déjà dedans)
- ✓ Itérer jusqu'à ne plus trouver d'items à ajouter

Etat de l'automate LR(0)

Les états de l'automate **LR(0)** sont les ensembles 8 d'items obtenus par fermeture (ensemble d'états LR(0), ou collection).
L'état initial $\mathbf{I_0}$ est obtenu par la fermeture de l'axiome (**S'→•S**).

Transitions de l'automate LR(0)

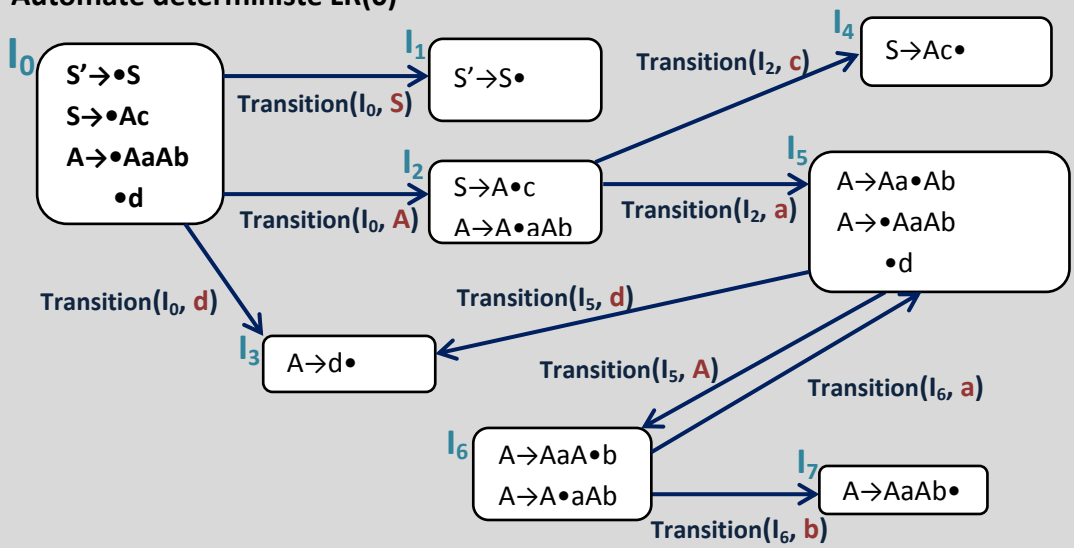
Soit **I** un ensemble d'items, et $X \in \{N \cup T\}^*$
On définit **Transition(I, X)** comme la **fermeture** de l'ensemble des items **A → αX•β**, tel que **A → α•Xβ** ∈ **I.**

Exemple :

Grammaire:

S' → S
S → Ac
A → AaAb | d

Automate déterministe LR(0)



3) Analyseur LR(0)

En fonction de l'automate **LR(0)** et de l'état de la pile, on va pouvoir déterminer quelle action effectuer, à savoir **REDUIRE** ou **LIRE**. Ces actions à effectuer seront inscrites dans la table **LR(0)**, qui se divise en deux parties :

- Table **ACTION** : 4 actions possibles (**Lire** / **Réduire** / **Accepter** / **Erreur**)
- Table **TRANSITION** : la partie de transition sur les non-terminaux.

Etats I_i	Terminaux	Non-Terminaux
	ACTIONS	TRANSITIONS

Remplissage de la table ACTION :

- Si $[S' \rightarrow S\bullet] \in I_i$ alors **ACTION**$[I_i, \$]$ = "**ACCEPTER**"
- Si $[X \rightarrow \beta\bullet] \in I_i$ alors **ACTION**$[I_i, a]$ = "**REDUIRE** par $X \rightarrow \beta$" $\forall a \in \{T \cup \$\}$
- Si $[X \rightarrow \alpha\bullet a\beta] \in I_i$ alors **ACTION**$[I_i, a]$ = "**LIRE + ALLER à l'état I_j**" où I_j = Transition(I_i, a) dans l'automate, $\forall a \in \{T \cup \$\}$
- "**ERREUR**" dans tous les autres cas.

Remplissage de la table TRANSITION :

Si dans l'automate Transition(I_i , X) = I_j alors **TRANSITION**$[I_i, X]$ = "I_j".

Reprise de l'exemple précédent :

	a	b	c	d	$	S'	S	A
I_0	erreur	erreur	erreur	I_3	erreur	erreur	I_1	I_2
I_1	erreur	erreur	erreur	erreur	ACCEPTER	erreur	erreur	erreur
I_2	I_5	erreur	I_4	erreur	erreur	erreur	erreur	erreur
I_3	A→d	A→d	A→d	A→d	A→d	erreur	erreur	erreur
I_4	S→Ac	S→Ac	S→Ac	S→Ac	S→Ac	erreur	erreur	erreur
I_5	erreur	erreur	erreur	I_3	erreur	erreur	erreur	I_6
I_6	I_5	I_7	erreur	erreur	erreur	erreur	erreur	erreur
I_7	A→AaAb	A→AaAb	A→AaAb	A→AaAb	A→AaAb	erreur	erreur	erreur

Dans la table **LR(0)**, quand un état entraîne une réduction (comme I_3, I_4 et I_7 dans l'exemple précédent), c'est toute la ligne qui est concernée (la réduction se fait pour n'importe quel terminal lu dans le mot).
La construction de l'automate **LR(0)** est indépendante du mot d'entrée. Il n'y a pas de symbole de pré-vision.

Remarque : pour tout état **I** de l'automate **LR(0)**, **I** ne contient au plus qu'une seule règle avec le • en fin de partie droite (pas de conflit réduction/réduction). Si plusieurs règles peuvent être placées dans la même case, il y a conflit, et cela signifie que la grammaire n'est pas **LR(0)**.

La pile est constituée de terminaux, non-terminaux et états I_i. Les états se trouvent toujours au sommet et à la base de la pile, ainsi qu'entre chaque terminal ou non-terminal. L'action à effectuer est lue depuis la table d'analyse, en prenant l'état présent en sommet de pile (à droite), et le caractère courant du mot. En cas de **lecture**, on ajoute le caractère courant en haut de la pile, puis on ajoute encore l'état indiqué dans la case correspondante dans la table. En cas de **réduction**, on remplace tous les terminaux et non-terminaux par le non-terminal de réduction (les états intermédiaires sont supprimés). Il reste à ajouter un état en sommet de pile : celui-ci se lit dans la table TRANSITION, dont la case est déterminée par l'état précédent le non-terminal de réduction dans la pile, et par ce non-terminal.

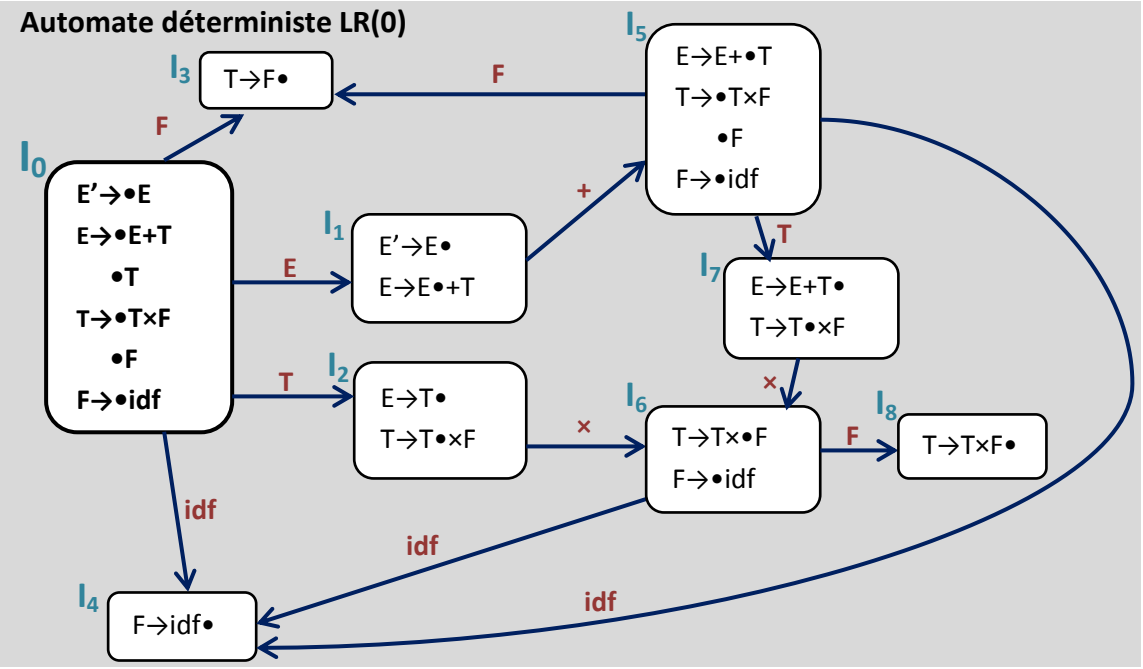
Exemple : déroulement complet de l'algorithme pour le mot dadbc$:			
Action n°	**Pile**	**Action**	**Mot**
0	I_0	*initialisation*	**dadbc$**
1	I_0 d I_3	Lecture + Aller à I_3	**adbc$**
2	I_0 A I_2	Réduction **A→d** Transition vers I_2	**adbc$**
3	I_0 A I_2 a I_5	Lecture + Aller à I_5	**dbc$**
4	I_0 A I_2 a I_5 d I_3	Lecture + Aller à I_3	**bc$**
5	I_0 A I_2 a I_5 A I_6	Réduction **A→d** Transition vers I_6	**bc$**
6	I_0 A I_2 a I_5 A I_6 b I_7	Lecture + Aller à I_7	**c$**
7	I_0 A I_2	Réduction **A→AaAb** Transition vers I_2	**c$**
8	I_0 A I_2 c I_4	Lecture + Aller à I_4	**$**
9	I_0 S I_1	Réduction **A→Ac** Transition vers I_1	**$**
10	I_0 S I_1	Analyse OK	**$**

4) Analyseur SLR(1)

Exemple :

Grammaire:

E' → E
E → E+T | T
T → T×F | F
F → idf



Nous allons voir que cette grammaire n'est pas **LR(0)**, car des conflits vont apparaître dans sa table d'analyse **LR(0)** :

	+	×	idf	$		E'	E	T	F
I_0	*erreur*	*erreur*	I_4	*erreur*		*err*	I_1	I_2	I_3
I_1	I_5	*erreur*	*erreur*	**ACCEPTER**		*err*	*erreur*	*erreur*	*erreur*
I_2	**E→T**	**E→T** *ou* I_6	**E→T**	**E→T**		*err*	*erreur*	*erreur*	*erreur*
I_3	**T→F**	**T→F**	**T→F**	**T→F**		*err*	*erreur*	*erreur*	*erreur*
I_4	**F→idf**	**F→idf**	**F→idf**	**F→idf**		*err*	*erreur*	*erreur*	*erreur*
I_5	*erreur*	*erreur*	I_4	*erreur*		*err*	*erreur*	I_7	I_3
I_6	*erreur*	*erreur*	I_4	*erreur*		*err*	*erreur*	*erreur*	I_8
I_7	**E→E+T**	**E→E+T** *ou* I_6	**E→E+T**	**E→E+T**		*err*	*erreur*	*erreur*	*erreur*
I_8	**T→T×F**	**T→T×F**	**T→T×F**	**T→T×F**		*err*	*erreur*	*erreur*	*erreur*

Déroulons l'algorithme pour le mot i+i×i$:

Action n°	Pile	Action	Mot
0	I_0	*initialisation*	**i+i×i$**
1	I_0 **i** I_4	Lecture + Aller à I_4	**+i×i$**
2	I_0 **F** I_3	Réduction **F→idf** Transition vers I_3	**+i×i$**
3	I_0 **T** I_2	Réduction **T→F** Transition vers I_2	**+i×i$**
4	I_0 **E** I_1	Réduction **E→T** Transition vers I_1	**+i×i$**
5	I_0 **E** I_1 **+** I_5	Lecture + Aller à I_5	**i×i$**
6	I_0 **E** I_1 **+** I_5 **i** I_4	Lecture + Aller à I_4	**×i$**
7	I_0 **E** I_1 **+** I_5 **F** I_3	Réduction **F→idf** Transition vers I_3	**×i$**
8	I_0 **E** I_1 **+** I_5 **T** I_7	Réduction **T→F** Transition vers I_7	**×i$**
9	I_0 **E** I_1 **+** I_5 **T** I_7	**CONFLIT** Réduction **E→E+T** **OU** Lecture du '×'	**×i$**

En se servant de la table **LR(0)**, il n'est pas possible de lire le mot **i+i×i** (à cause du conflit dans la case **[I_7,×]**). La bonne action aurait été ici de lire le caractère suivant (et non de réduire).

Comment résoudre ce problème ?

On résout le conflit lecture/réduction en regardant si le terminal présent en tête du texte restant à analyser était dans l'ensemble des suivants du non-terminal en partie gauche de la production utilisable pour la réduction.

Ceci est formalisé dans l'analyseur **SLR(1)** : la table d'analyse **SLR(1)** se remplit de la même façon que la table d'analyse **LR(0)**, à la seule différence qu'**on y ajoute une condition** en ce qui concerne le remplissage de la table **ACTION :**

Si [X→β•] ∈ I_i, X≠S' (*X n'est pas l'axiome*), alors **action[I_i, a] = "REDUIRE par X→β"** **pour tout a ∈ Suivant(X)**.

Cette condition étant maintenant prise en compte, nous pouvons construire la table d'analyse **SLR(1)**.

Sachant que : Suivants(**E**) = **$, +**
Suivants(**T**) = **$, +, ×**
Suivants(**F**) = **$, +, ×**

	+	×	idf	$		E'	E	T	F
I_0	*erreur*	*erreur*	I_4	*erreur*		*err*	I_1	I_2	I_3
I_1	I_5	*erreur*	*erreur*	**ACCEPTER**		*err*	*erreur*	*erreur*	*erreur*
I_2	**E→T** *car + suivant de E*	I_6	*erreur*	**E→T** *car $ suivant de E*		*err*	*erreur*	*erreur*	*erreur*
I_3	**T→F** *car + suivant de T*	**T→F** *car × suivant de T*	*erreur*	**T→F** *car $ suivant de T*		*err*	*erreur*	*erreur*	*erreur*
I_4	**F→idf** *car + suivant de F*	**F→idf** *car × suivant de F*	*erreur*	**F→idf** *car $ suivant de F*		*err*	*erreur*	*erreur*	*erreur*
I_5	*erreur*	*erreur*	I_4	*erreur*		*err*	*erreur*	I_7	I_3
I_6	*erreur*	*erreur*	I_4	*erreur*		*err*	*erreur*	*erreur*	I_8
I_7	**E→E+T** *car + suivant de E*	I_6	*erreur*	**E→E+T** *car $ suivant de E*		*err*	*erreur*	*erreur*	*erreur*
I_8	**T→T×F** *car + suivant de T*	**T→T×F** *car × suivant de T*	*erreur*	**T→T×F** *car $ suivant de T*		*err*	*erreur*	*erreur*	*erreur*

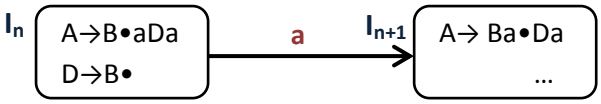
Déroulons à présent l'algorithme, toujours pour le mot i+i×i$:

Action n°	Pile	Action	Mot
0	I_0	*initialisation*	**i+i×i$**
1	I_0 i I_4	Lecture + Aller à I_4	**+i×i$**
2	I_0 F I_3	Réduction **F→idf** Transition vers I_3	**+i×i$**
3	I_0 T I_2	Réduction **T→F** Transition vers I_2	**+i×i$**
4	I_0 E I_1	Réduction **E→T** Transition vers I_1	**+i×i$**
5	I_0 E I_1 + I_5	Lecture + Aller à I_5	**i×i$**
6	I_0 E I_1 + I_5 i I_4	Lecture + Aller à I_4	**×i$**
7	I_0 E I_1 + I_5 F I_3	Réduction **F→idf** Transition vers I_3	**×i$**
8	I_0 E I_1 + I_5 T I_7	Réduction **T→F** Transition vers I_7	**×i$**
9	I_0 E I_1 + I_5 T I_7 × I_6	Lecture + Aller à I_6	**i$**
10	I_0 E I_1 + I_5 T I_7 × I_6 i I_4	Lecture + Aller à I_4	**$**
11	I_0 E I_1 + I_5 T I_7 × I_6 F I_8	Réduction **F→idf** Transition vers I_8	**$**
12	I_0 E I_1 + I_5 T I_7	Réduction **T→T×F** Transition vers I_7	**$**
13	I_0 E I_1	Réduction **E→E+T** Transition vers I_1	**$**
14	I_0 E I_1	**Analyse OK**	**$**

Remarque : le fonctionnement de cet algorithme fait qu'on ne réduit jamais jusqu'à l'axiome de la grammaire augmentée (par exemple ici, on ne termine pas par la réduction **E'→E**). On s'arrête toujours à l'axiome de la grammaire non augmentée (**E**). Cela était déjà le cas pour l'analyse **LR(0)**.

5) Analyseur LR(1)

Dans certains cas, l'analyseur **SLR(1)** n'est pas assez puissant pour se rappeler assez de « contexte » pour décider de l'action à choisir. Il peut donc entraîner, comme l'analyseur **LR(0)**, des conflits lecture/réduction.



Si le terminal **a** appartient aux suivants de **D**, alors on aura un conflit dans la case **ACTION[I_n, a]** : soit réduire par **D→B**, soit lire le caractère et passer à l'état **In+1**.

La condition ajoutée à l'analyseur **SLR(1)** n'est dans ce cas pas suffisante.

La solution à ce problème est fournie par l'analyseur **LR(1)** : on va rajouter de l'information supplémentaire dans les items. Cette information supplémentaire est appelée **contexte** (ou **symbole de prévision**), et correspondra aux terminaux pour lesquels on fera une réduction.

Un item **LR(1)** devient **[A→α•β, [a]]**, où **a** $\in T \cup \$$.

Le symbole de prévision n'aura d'effet que sur les items de réduction. Pour **[A→α•β, [a]]**, l'action sera **"réduire par A→α uniquement par a".**

Pour résumer :

- Dans l'analyse **LR(0)**, pour un état donné dont un item est en fin de production, on effectuait la réduction pour tous les terminaux.
- Dans l'analyse **SLR(1)**, on réduisait uniquement pour les terminaux appartenant aux suivants du non-terminal.
- Dans l'analyse **LR(1)**, on ne réduit plus que pour les terminaux appartenant au contexte, le but étant de minimiser les risques de conflit.

Définition et construction des items LR(1)

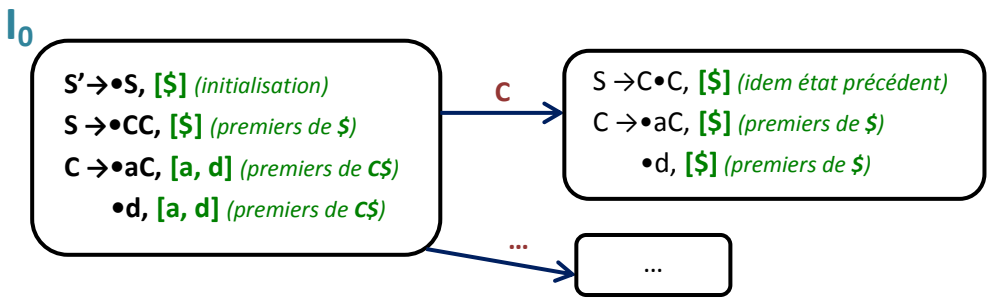
On se sert toujours de la grammaire augmentée. L'automate se construit de la même façon, en rajoutant les contextes associés à chaque item. L'état initial I_0 correspond à la fermeture de **(S'→•S, [$])**.

Calcul de la fermeture de **I** :
Si **[A→α•Bβ, [a]]** est dans **I** (avec **B→γ**) :
alors, pour tout terminal **b** appartenant aux premiers de (**βa**), ajouter **[B→•γ, [b]]** à **Fermeture(I).**

Exemple :

Grammaire:

S' → S
S → CC
C → aC | d



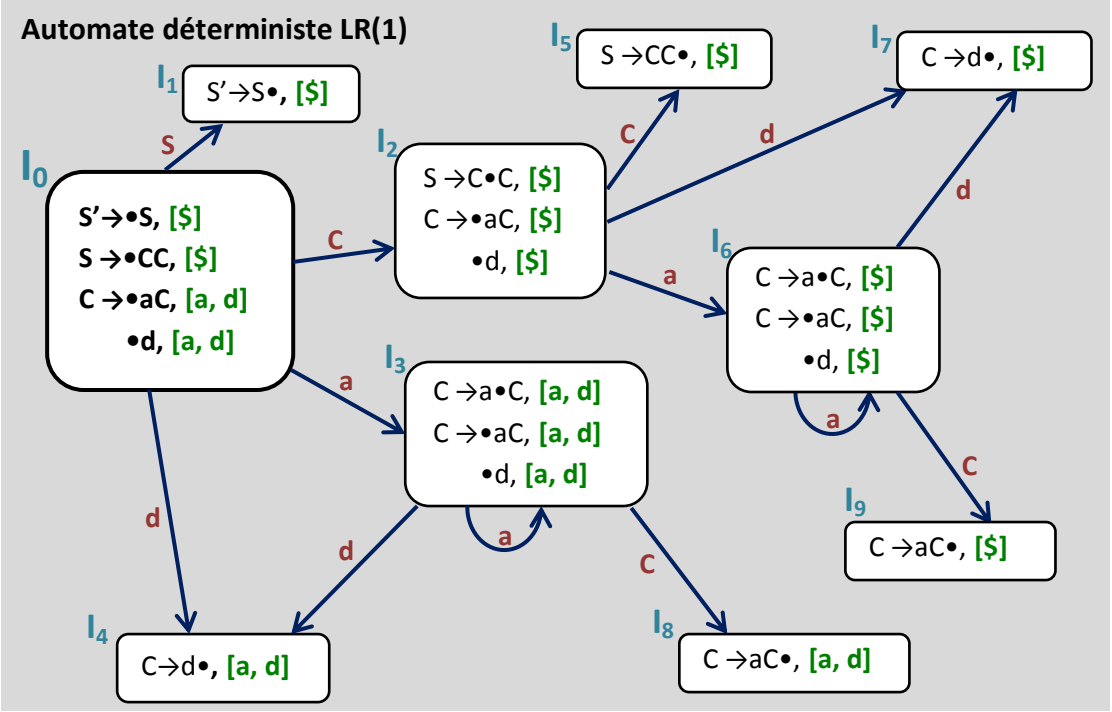
L'automate **LR(1)** se construit de la même façon que les automates **LR(0)** et **SLR(1)**, mais avec des items de type **LR(1)**. On ne calcule pas le contexte dans une transition (le contexte reste le même qu'à l'état précédent).

Grammaire:

S' → S
S → CC
C → aC | d

Les états I_4 et I_7 ont même noyau (seul leur contexte change).

C'est également le cas des états I_3 et I_6, ou encore des états I_8 et I_9.



Construction de la table LR(1) :

Remplissage de la table ACTION :

- Si [S'→S•] ∈ I_i alors **ACTION**[I_i, $] = "**ACCEPTER**"
- Si [X→β•, **[a]**] ∈ I_i alors **ACTION**[I_i, **a**] = "**REDUIRE** par X→β"
- Si [X→α•aβ, **[b]**] ∈ I_i alors **ACTION**[I_i, a] = "**LIRE + ALLER à l'état I_j**"
  où I_j = Transition(I_i , a) dans l'automate, ∀a,b ∈ {T ∪ $ }
- "**ERREUR**" dans tous les autres cas.

Remplissage de la table TRANSITION :

Si dans l'automate Transition(I_i , X) = I_j
alors **TRANSITION**[I_i, X] = "I_j".

Reprise de l'exemple précédent : construction de la table LR(1).

	a	d	$		S'	S	C
I_0	I_3	I_4	*erreur*		*err*	I_1	I_2
I_1	*erreur*	*erreur*	**ACCEPTER**		*err*	*erreur*	*erreur*
I_2	I_6	I_7	*erreur*		*err*	*erreur*	I_5
I_3	I_3	I_4	*erreur*		*err*	*erreur*	I_8
I_4	**C→d**	**C→d**	*erreur*		*err*	*erreur*	*erreur*
I_5	*erreur*	*erreur*	**S→CC**		*err*	*erreur*	*erreur*
I_6	I_6	I_7	*erreur*		*err*	*erreur*	I_9
I_7	*erreur*	*erreur*	**C→d**		*err*	*erreur*	*erreur*
I_8	**C→aC**	**C→aC**	*erreur*		*err*	*erreur*	*erreur*
I_9	*erreur*	*erreur*	**C→aC**		*err*	*erreur*	*erreur*

Remarque : la grammaire utilisée en exemple était **LR(0)**.

6) Analyse LALR(1)

L'analyse **LALR(1)** consiste à rassembler les états ayant même noyau pour obtenir un automate et une table **LR(0)** sans introduire de conflits, quand cela est possible. En effet, l'analyse **LR(1)** génère un grand nombre d'états par rapport à l'analyse **SLR(1)**. Ces états peuvent bien souvent être fusionnés, afin d'optimiser l'analyseur.

Reprise de l'exemple précédent : on fusionne les états I_3 / I_6, les états I_4 / I_7 et les états I_8 / I_9.

	a	d	$		S'	S	C
I_0	I_3 / I_6	I_4 / I_7	erreur		err	I_1	I_2
I_1	erreur	erreur	ACCEPTER		err	erreur	erreur
I_2	I_3 / I_6	I_4 / I_7	erreur		err	erreur	I_5
I_3 / I_6	I_3 / I_6	I_4 / I_7	erreur		err	erreur	I_8 / I_9
I_4 / I_7	C→d	C→d	C→d		err	erreur	erreur
I_5	erreur	erreur	S→CC		err	erreur	erreur
I_8 / I_9	C→aC	C→aC	C→aC		err	erreur	erreur

Dans cet exemple, tous les états ayant même noyau ont pu être fusionnés.

7) Analyse ascendante et grammaire ambigüe

Nous allons traiter le cas d'une grammaire ambigüe (grammaire pour laquelle un même mot peut être interprété différemment, et donner deux arbres syntaxiques différents). Pour cela, nous allons construire l'automate **LR(0)** puis la table d'analyse **SLR(1)** sur la grammaire suivante :

Grammaire:

E' → E

E → EOE

entier

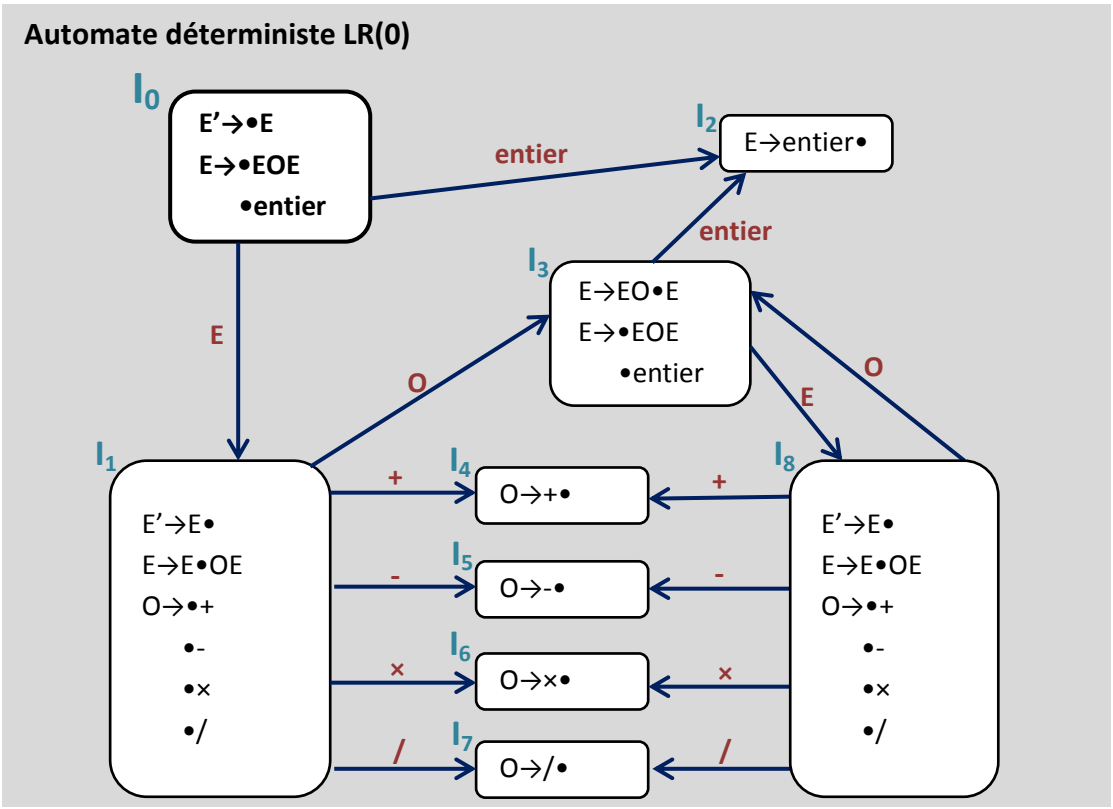
O → +

-

×

/

Automate déterministe LR(0)



Sachant que : Suivants(**O**) = **entier**
Suivants(**E**) = **+**, **-**, **×**, **/**

	entier	+	-	×	/	$		E'	E	O
I_0	I_2	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*		*err*	I_1	*erreur*
I_1	*erreur*	I_4	I_5	I_6	I_7	**ACCEPTER**		*err*	*erreur*	I_3
I_2	*erreur*	**E→entier**	**E→entier**	**E→entier**	**E→entier**	*erreur*		*err*	*erreur*	*erreur*
I_3	I_2	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*		*err*	I_8	*erreur*
I_4	**O→+**	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*		*err*	*erreur*	*erreur*
I_5	**O→-**	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*		*err*	*erreur*	*erreur*
I_6	**O→×**	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*		*err*	*erreur*	*erreur*
I_7	**O→/**	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*		*err*	*erreur*	*erreur*
I_8	*erreur*	**E→EOE** *ou* I_4	**E→EOE** *ou* I_5	**E→EOE** *ou* I_6	**E→EOE** *ou* I_7	**E→EOE**		*err*	*erreur*	I_3

*Des **conflits** sont présents dans la table d'analyse **SLR(1)** : la grammaire n'était donc pas **SLR(1)**.*
*Selon le ou les symbole(s) trouvé(s), il va falloir faire soit **SHIFT**, soit **REDUCE**.*

Déroulons l'algorithme pour le mot 1-3×5$:

Action n°	Pile	Action	Mot
0	I_0	*initialisation*	**1-3×5$**
1	I_0 1 I_2	Lecture + Aller à I_2	1**-3×5$**
2	I_0 E I_1	Réduction **E→entier** Transition vers I_1	1**-3×5$**
3	I_0 E I_1 - I_5	Lecture + Aller à I_5	1-**3×5$**
4	I_0 E I_1 O I_3	Réduction **O→-** Transition vers I_3	1-**3×5$**
5	I_0 E I_1 O I_3 3 I_2	Lecture + Aller à I_2	1-3**×5$**
6	I_0 E I_1 O I_3 E I_8	Réduction **E→entier** Transition vers I_8	1-3**×5$**

A partir d'ici, on peut **soit lire** le symbole suivant et passer à l'état I_6, **soit réduire par E→EOE**.

Comme la multiplication est prioritaire sur l'addition, la bonne solution est ici de **lire** le caractère suivant. Réduire signifierait donner implicitement priorité à la soustraction, c'est-à-dire entourer de parenthèses la première opération : **(1-3)×5**, ce qui fausserait le résultat.
En revanche, si le mot était **1-3+5**, on aurait pu réduire, l'addition n'étant pas prioritaire sur la soustraction.
Il faut donc ajouter des priorités aux opérateurs de la grammaire afin de lever les conflits : **priorité(×) > priorité(+)**.

Rendre l'analyseur déterministe

On suppose que la pile, en arrivant à un second opérateur, est constituée des éléments suivants :

I_0 E I_1 O I_3 E I_8

Quel comportement faut-il adopter, en arrivant à ce second opérateur (caractère suivant), en fonction du premier opérateur (c'est-à-dire le non-terminal **O** dans la pile) ?

Ici, réduire par **E→EOE** revient à placer implicitement des parenthèses autour de la partie réduite : dans l'exemple, cela donnerait **(1-3)×5**, ce qui est faux mathématiquement. Lire revient donc à retarder le moment où on réduira, et donc à ne pas forcément donner priorité à la partie gauche du calcul.

En revanche, dans le cas où les deux opérateurs ont la même priorité, on privilégie l'associativité gauche, c'est-à-dire qu'on préfère placer implicitement des parenthèses autour de la partie gauche du calcul. En effet, cette règle se révèle importante dans le cas où par exemple, les deux opérateurs sont des soustractions ou des divisions :

1-3-5 est égal à **(1-3)-5**, mais est différent de **1-(3-5)** !

Tableau récapitulatif, pour notre exemple :

	Premier opérateur (O) **de basse priorité (+ ou -)**	**Premier opérateur** (O) **de haute priorité (× ou /)**
Second opérateur (caractère suivant) **de basse priorité (+ ou -)**	Exemple : **1-3+5** Il faut **réduire** : **(1-3)+5** car on privilégie l'associativité gauche.	Exemple : **1×3+5** Il faut **réduire** : **(1×3)+5** car priorité du **×** sur le **+**.
Second opérateur (caractère suivant) **de haute priorité (× ou /)**	Exemple : **1-3×5** Il faut **lire** : **1-(3×5)** car priorité du **×** sur le **-**.	Exemple : **1×3×5** Il faut **réduire** : **(1×3)×5** car on privilégie l'associativité gauche.

Pour que l'analyseur syntaxique fonctionne tel qu'on le souhaite, on crée des **sous-états** à l'état $\mathbf{I_8}$:

8.1 E→E+E• **8.2** E→E×E• **8.3** E→E-E• **8.4** E→E/E•

C'est le premier opérateur (**O**) qui détermine dans lequel de ces sous-états on va entrer. En fonction du second opérateur, on détermine l'action à effectuer :

- Si on est à l'**état 8.1 ou 8.3**, et si le second opérateur est :
  - **+** ou **-** : il faut réduire (associativité gauche)
  - **×** ou **/** : il faut lire (priorité du second opérateur)
- Si on est à l'**état 8.2 ou 8.4**, et si le second opérateur est :
  - **+** ou **-** : il faut réduire (priorité du premier opérateur)
  - **×** ou **/** : il faut réduire (associativité gauche)

Remarques générales

Pour supprimer les conflits lecture/réduction dus à l'ambiguïté d'une grammaire, on peut soit transformer la grammaire afin d'y supprimer son ambiguïté, soit la conserver, et dans ce cas utiliser les **précédences** (de l'anglais *operator precedence*, priorité de l'opérateur).

Si on a deux opérateurs **u** puis **t** définis de cette manière dans un des états de l'automate **LR(1)**:

R→α•**t**β , […]
Q→α**u****R**•, [**t**]

*Si on est à cet état et que le caractère suivant est **t** :*
*on fait soit un **shift**, soit un **reduce** (conflit).*

Si des **priorités** (ou **précédences**) sont placées sur les opérateurs **t** et **u**, alors :

- Si priorité(**u**) **>** priorité(**t**), alors il faut faire un **REDUCE**.
- Si priorité(**u**) **<** priorité(**t**), alors il faut faire un **SHIFT**.
- Si priorité(**u**) **=** priorité(**t**), alors peu importe. En général, les analyseurs choisissent **SHIFT**.

8) Fonctions sémantiques en analyse syntaxique ascendante

Soit la grammaire:

A' → A

A → V
(A.A)
(AS)

S → ,AS
ε

V → entier
nil

La grammaire ci-contre correspond à une notation utilisée pour décrire les arbres binaires. On note **(A.B)** l'arbre binaire composé des sous-arbres **A** et **B**. Si un arbre est réduit à une feuille, il est noté **nil.**

La notation gère également les listes : on note **(A, B, …, E)** la liste contenant les éléments **A**, **B**, … et **E** (successivement). Les éléments peuvent être des listes ou des arbres et donc aussi avoir des valeurs entières, ou bien peuvent être égaux à **nil**.

Exemple : **((3.2) , nil , nil, 4)** est une liste (dont la notation respecte la syntaxe décrite par la grammaire ci-contre).

Les listes suivantes sont également valides : **nil**
13
(4)
(1, (2.3), 4)

Problème : on souhaite avoir une **représentation commune** aux arbres et aux listes.

On décide que la liste **(A, B, …, E)** est représentée par l'arbre **(A.(B.(… .(E.nil) …)))** *(Notation pointée uniquement)*.

Pour cela, on va construire un analyseur syntaxique ascendant pour cette grammaire, sur lequel on va ajouter des fonctions sémantiques permettant de transformer la première notation en notation pointée uniquement. Ces petites fonctions sémantiques seront chacune associée à un type de réduction (par exemple, à **A→V**, ou **V → nil**). Chaque fonction sera exécutée dès que sa réduction associée se produira, lors de l'analyse d'un mot. En revanche, les fonctions ne sont pas exécutées lorsqu'on effectue une lecture.

On dispose de la table d'analyse **SLR(1)** de la grammaire *(on suppose que l'automate **LR(0)** a déjà été construit)* :

	(	)	.	,	entier	nil	$		A'	A	V	S
I_0	I_3	erreur	erreur	erreur	I_4	I_5	erreur		err	I_1	I_2	erreur
I_1	erreur	erreur	erreur	erreur	erreur	erreur	ACCEPTER		err	erreur	erreur	erreur
I_2	erreur	A→V	A→V	A→V	erreur	erreur	A→V		err	erreur	erreur	erreur
I_3	I_3	erreur	erreur	erreur	I_4	I_5	erreur		err	I_6	I_2	erreur
I_4	erreur	V→entier	V→entier	V→entier	erreur	erreur	V→entier		err	erreur	erreur	erreur
I_5	erreur	V→nil	V→nil	V→nil	erreur	erreur	V→nil		err	erreur	erreur	erreur
I_6	erreur	S→ε	I_7	I_9	erreur	erreur	erreur		err	erreur	I_2	I_8
I_7	I_3	erreur	erreur	erreur	I_4	I_5	erreur		err	I_{10}	I_2	erreur
I_8	erreur	I_{11}	erreur	erreur	erreur	erreur	erreur		err	erreur	erreur	erreur
I_9	I_3	erreur	erreur	erreur	I_4	I_5	erreur		err	I_{12}	I_2	erreur
I_{10}	erreur	I_{13}	erreur	erreur	erreur	erreur	erreur		err	erreur	erreur	erreur
I_{11}	erreur	A→(AS)	A→(AS)	A→(AS)	erreur	erreur	A→(AS)		err	erreur	erreur	erreur
I_{12}	erreur	S→ε	erreur	I_9	erreur	erreur	erreur		err	erreur	erreur	I_{14}
I_{13}	erreur	A→(A.A)	A→(A.A)	A→(A.A)	erreur	erreur	A→(A.A)		err	erreur	erreur	erreur
I_{14}	erreur	S→,AS	erreur	erreur	erreur	erreur	erreur		err	erreur	erreur	erreur

On va traiter le mot suivant : **(1, (2.3), 4)**. Il s'agit bien d'une liste, contenant des éléments de type entier et arbre. Sa notation pointée est la suivante : **(1.((2.3).(4.nil)))**

L'algorithme d'analyse syntaxique combiné aux fonctions sémantiques va nous permettre d'obtenir cette notation, sous la forme d'une chaîne de caractères en sortie. En plus de la pile contenant les états, terminaux et non-terminaux, qui est habituellement utilisée dans cet algorithme, on va se servir d'une **seconde pile** contenant des **chaînes de caractères**. Elle sera utilisée par les fonctions sémantiques et permettra de mettre de côté des « morceaux » de l'affichage final.

Dans notre exemple, **cinq fonctions sémantiques** seront nécessaires pour générer la notation pointée :

Fonction F_0 :
```
afficher(sommetPile)
```

Fonction F_1 :
```
empiler(entier)
```

Fonction F_2 :
```
empiler("nil")
```

Fonction F_3 :
```
x1 := depiler()
x2 := depiler()
empiler("(" x1 "." x2 ")")
```

Fonction F_4 :
```
x := depiler()
empiler("(" x ".nil)")
```

- En cas de **mot ACCEPTE** (*ou réduction* **A' → A**) : exécuter la **fonction F_0**
- En cas de réduction **V→ entier** : exécuter la **fonction F_1**
- En cas de réduction **V → nil** : exécuter la **fonction F_2**
- En cas de réduction **A → (A.A)** ou en cas de réduction **S → ,AS** : exécuter la **fonction F_3**
- En cas de réduction **S → ε** : exécuter la **fonction F_4**

Pour comprendre le fonctionnement de ces fonctions sémantiques, déroulons l'algorithme pour **(1, (2.3), 4)$** :

n°	Pile de l'analyseur	Action	Mot	Pile *(fonc. sem.)*
0	I_0	*initialisation*	**(1, (2.3), 4)$**	*(vide)*
1	I_0 (I_3	Lecture + Aller à I_3	**(1, (2.3), 4)$**	*(vide)*
2	I_0 (I_3 1 I_4	Lecture + Aller à I_4	**(1, (2.3), 4)$**	*(vide)*
3	I_0 (I_3 V I_2	Réduct. **V→entier** Transition vers I_2	**(1, (2.3), 4)$**	"1" Fonction F_1()
4	I_0 (I_3 A I_6	Réduction **A→V** Transition vers I_6	**(1, (2.3), 4)$**	"1"
5	I_0 (I_3 A I_6 , I_9	Lecture + Aller à I_9	**(1, (2.3), 4)$**	"1"
6	I_0 (I_3 A I_6 , I_9 (I_3	Lecture + Aller à I_3	**(1, (2.3), 4)$**	"1"
7	I_0 (I_3 A I_6 , I_9 (I_3 2 I_4	Lecture + Aller à I_4	**(1, (2.3), 4)$**	"1"
8	I_0 (I_3 A I_6 , I_9 (I_3 V I_2	Réduct. **V→entier** Transition vers I_2	**(1, (2.3), 4)$**	"2" "1" Fonction F_1()
9	I_0 (I_3 A I_6 , I_9 (I_3 A I_6	Réduction **A→V** Transition vers I_6	**(1, (2.3), 4)$**	"2" "1"
10	I_0 (I_3 A I_6 , I_9 (I_3 A I_6 . I_7	Lecture + Aller à I_7	**(1, (2.3), 4)$**	"2" "1"

n°	Pile de l'analyseur	Action	Mot	Pile *(fonc. sem.)*
11	I_0 (I_3 A I_6 , I_9 (I_3 A I_6 . I_7 3 I_4	Lecture + Aller à I_4	(1, (2.3), 4)$	"2" "1"
12	I_0 (I_3 A I_6 , I_9 (I_3 A I_6 . I_7 V I_2	Réduct. **V→entier** Transition vers I_2	(1, (2.3), 4)$	"3" "2" "1" Fonction F_1()
13	I_0 (I_3 A I_6 , I_9 (I_3 A I_6 . I_7 A I_{10}	Réduction **A→V** Transition vers I_{10}	(1, (2.3), 4)$	"3" "2" "1"
14	I_0 (I_3 A I_6 , I_9 (I_3 A I_6 . I_7 A I_{10}) I_{13}	Lecture + Aller à I_{13}	(1, (2.3), 4)$	"3" "2" "1"
15	I_0 (I_3 A I_6 , I_9 A I_{12}	Réduct. **A→(A.A)** Transition vers I_{12}	(1, (2.3), 4)$	"(2.3)" "1" Fonction F_3()
16	I_0 (I_3 A I_6 , I_9 A I_{12} , I_9	Lecture + Aller à I_9	(1, (2.3), 4)$	"(2.3)" "1"
17	I_0 (I_3 A I_6 , I_9 A I_{12} , I_9 4 I_4	Lecture + Aller à I_4	(1, (2.3), 4)$	"(2.3)" "1"
18	I_0 (I_3 A I_6 , I_9 A I_{12} , I_9 V I_2	Réduct. **V→entier** Transition vers I_2	(1, (2.3), 4)$	"4" "(2.3)" "1" Fonction F_1()
19	I_0 (I_3 A I_6 , I_9 A I_{12} , I_9 A I_{12}	Réduction **A→V** Transition vers I_{12}	(1, (2.3), 4)$	"4" "(2.3)" "1"
20	I_0 (I_3 A I_6 , I_9 A I_{12} , I_9 A I_{12} S I_{14}	Réduction **S→ε** Transition vers I_{14}	(1, (2.3), 4)$	"(4.nil)" "(2.3)" "1" Fonction F_4()
21	I_0 (I_3 A I_6 , I_9 A I_{12} S I_{14}	Réduction **S→,AS** Transition vers I_{12}	(1, (2.3), 4)$	"((2.3).(4.nil))" "1" Fonction F_3()
22	I_0 (I_3 A I_6 S I_8	Réduction **S→,AS** Transition vers I_8	(1, (2.3), 4)$	"(1.((2.3).(4.nil)))" Fonction F_3()
23	I_0 (I_3 A I_6 S I_8) I_{11}	Lecture + Aller à I_{11}	(1, (2.3), 4)$	"(1.((2.3).(4.nil)))"
24	I_0 A I_1	Réduction **A→(AS)** Transition vers I_1	(1, (2.3), 4)$	"(1.((2.3).(4.nil)))"
25	I_0 A I_1	**Analyse OK**	(1, (2.3), 4)$	"(1.((2.3).(4.nil)))" Fonction F_0() Ecriture du sommet de pile : **(1.((2.3).(4.nil)))**

L'analyseur syntaxique accepte le mot qui était correct, et exécute la fonction F_0, à savoir l'affichage du sommet de la pile. Ainsi, on obtient bien en sortie la **notation pointée** du mot passé en entrée de l'analyseur.

Conclusion

L'analyse descendante vue en cours de ***MI2*** est une méthode d'analyse syntaxique rapide à mettre en place, mais relativement limitée : la grammaire doit être **LL(1)** ; la récursivité gauche est interdite.

L'analyse ascendante est plus complexe à mettre en œuvre, mais bien plus puissante, puisqu'un analyseur ascendant est capable de traiter des grammaires **LR(1)** ou **LL(1)**, récursives, et même ambigües.