

Université Paris-Sud

Compilation et langages

Thibaut Balabonski
d'après Jean-Christophe Filliâtre @ ENS

Cours 3 / 30 septembre 2016

Analyse **syntaxique** et conflits

L'objectif de l'analyse syntaxique est de reconnaître les phrases appartenant à la syntaxe du langage

Son entrée est le flot des lexèmes construits par l'analyse lexicale, sa sortie est un arbre de syntaxe abstraite

suite de lexèmes

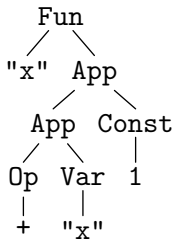
`fun` `x` `->` `(` `x` `+` `1` `)`



analyse syntaxique



syntaxe abstraite



En particulier, l'analyse syntaxique doit détecter les erreurs de syntaxe et

- les localiser précisément
- les identifier (le plus souvent seulement « erreur de syntaxe » mais aussi « parenthèse non fermée », etc.)
- voire, reprendre l'analyse pour découvrir de nouvelles erreurs

Pour l'analyse syntaxique, on va utiliser

- une **grammaire non contextuelle** pour décrire la syntaxe
- un **automate à pile** pour la reconnaître

C'est l'analogue des expressions régulières / automates finis utilisés dans l'analyse lexicale

Définition

Une grammaire non contextuelle (ou hors contexte) est un quadruplet (N, T, S, R) où

- *N est un ensemble fini de **symboles non terminaux***
- *T est un ensemble fini de **symboles terminaux***
- *$S \in N$ est le symbole de départ (dit **axiome**)*
- *$R \subseteq N \times (N \cup T)^*$ est un ensemble fini de **règles de production***

Exemple : expressions arithmétiques

- Symboles terminaux : +, *, (,), int
- Symboles non terminaux : E
- Symbole de départ : E
- Règles :
$$\begin{array}{lcl} E & \rightarrow & E + E \\ & & | \quad E * E \\ & & | \quad (E) \\ & & | \quad \text{int} \end{array}$$

int désigne ici le lexème correspondant à une constante entière (*i.e.* sa nature, pas sa valeur)

Définition

Un mot $u \in (N \cup T)^*$ se **dérive** en un mot $v \in (N \cup T)^*$, et on note $u \rightarrow v$, s'il existe une décomposition

$$u = u_1 X u_2$$

avec $X \in N$, $X \rightarrow \beta \in R$ et

$$v = u_1 \beta u_2$$

Exemple :

$$\underbrace{E *}_{u_1} \left(\underbrace{E}_X \right) \underbrace{)}_{u_2} \rightarrow E * \left(\underbrace{E + E}_{\beta} \right)$$

Une suite $w_1 \rightarrow w_2 \rightarrow \dots \rightarrow w_n$ est appelée une dérivation

On parle de **dérivation gauche** (resp. **droite**) si le non terminal réduit est systématiquement le plus à gauche *i.e.* $u_1 \in T^*$ (resp. le plus à droite *i.e.* $u_2 \in T^*$)

On note \rightarrow^* la clôture réflexive transitive de \rightarrow

$$\begin{aligned} E &\rightarrow E * E \\ &\rightarrow \text{int} * E \\ &\rightarrow \text{int} * (E) \\ &\rightarrow \text{int} * (E + E) \\ &\rightarrow \text{int} * (\text{int} + E) \\ &\rightarrow \text{int} * (\text{int} + \text{int}) \end{aligned}$$

En particulier

$$E \rightarrow^* \text{int} * (\text{int} + \text{int})$$

Définition

À toute dérivation $S \rightarrow^* w$, on peut associer un **arbre de dérivation**, dont les nœuds sont étiquetés ainsi

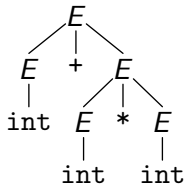
- la racine est S
- les feuilles forment le mot w dans l'ordre infixe
- tout nœud interne X est un non terminal dont les fils sont étiquetés par $\beta \in (N \cup T)^*$ avec $X \rightarrow \beta$ une règle de la dérivation

Attention : ce n'est pas la même chose que l'arbre de syntaxe abstraite

La dérivation gauche

$$E \rightarrow E + E \rightarrow \text{int} + E \rightarrow \text{int} + E * E \rightarrow \text{int} + \text{int} * E \rightarrow \text{int} + \text{int} * \text{int}$$

donne l'arbre de dérivation



mais la dérivation droite

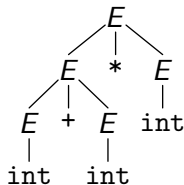
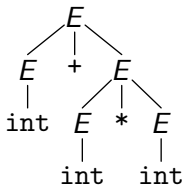
$$E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * \text{int} \rightarrow E + \text{int} * \text{int} \rightarrow \text{int} + \text{int} * \text{int}$$

également

Définition

Une grammaire est dite **ambiguë** si un mot au moins admet plusieurs arbres de dérivation

Exemple : le mot `int + int * int` admet les deux arbres de dérivations

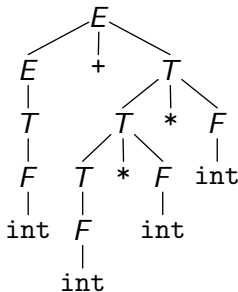


Pour ce langage-là, il est néanmoins possible de proposer une autre grammaire, non ambiguë, qui définit le même langage

$$\begin{array}{lcl} E & \rightarrow & E + T \\ & | & T \\ T & \rightarrow & T * F \\ & | & F \\ F & \rightarrow & (E) \\ & | & \text{int} \end{array}$$

Cette nouvelle grammaire traduit la priorité de la multiplication sur l'addition, et le choix d'une associativité à gauche pour ces deux opérations

Ainsi, le mot `int + int * int * int` n'a plus qu'un seul arbre de dérivation, à savoir



correspondant à la dérivation gauche

$$\begin{aligned} E &\rightarrow E + T \rightarrow T + T \rightarrow F + T \rightarrow \text{int} + T \rightarrow \text{int} + T * F \\ &\rightarrow \text{int} + T * F * F \rightarrow \text{int} + F * F * F \rightarrow \text{int} + \text{int} * F * F \\ &\rightarrow \text{int} + \text{int} * \text{int} * F \rightarrow \text{int} + \text{int} * \text{int} * \text{int} \end{aligned}$$

Déterminer si une grammaire est ou non ambiguë n'est **pas décidable**

On va utiliser des **critères décidables suffisants** pour garantir qu'une grammaire est non ambiguë, et pour lesquels on sait en outre décider l'appartenance au langage efficacement (avec un automate à pile déterministe)

Les classes de grammaires définies par ces critères s'appellent LL(1), LR(0), SLR(1), LALR(1), LR(1), etc.

Analyse ascendante

L'idée est de lire l'entrée de gauche à droite en essayant de reconnaître des membres droits de productions pour construire l'arbre de dérivation de bas en haut (*bottom-up parsing*)

L'analyse manipule une pile qui est un mot de $(T \cup N)^*$

À chaque instant, deux actions sont possibles

- opération de **lecture** (*shift* en anglais) :
on lit un terminal de l'entrée et on l'empile
- opération de **réduction** (*reduce* en anglais) :
on reconnaît en sommet de pile le membre droit β d'une production $X \rightarrow \beta$, et on remplace β par X en sommet de pile

Dans l'état initial, la pile est vide

Lorsqu'il n'y a plus d'action possible, l'entrée est reconnue si elle a été entièrement lue et si la pile est réduite à S

	pile	entrée	action
	ϵ	int+int*int	lecture
	int	+int*int	réduction $F \rightarrow \text{int}$
	F	+int*int	réduction $T \rightarrow F$
	T	+int*int	réduction $E \rightarrow T$
$E \rightarrow E + T$	E	+int*int	lecture
T	$E+$	int*int	lecture
$T \rightarrow T * F$	$E+\text{int}$	*int	réduction $F \rightarrow \text{int}$
F	$E+F$	*int	réduction $T \rightarrow F$
$F \rightarrow (E)$	$E+T$	*int	lecture
int	$E+T*$	int	lecture
	$E+T*\text{int}$		réduction $F \rightarrow \text{int}$
	$E+T*F$		réduction $T \rightarrow T*F$
	$E+T$		réduction $E \rightarrow E+T$
	E		succès

Comment prendre la décision lecture / réduction ?

En se servant d'un automate fini et en examinant les k premiers caractères de l'entrée ; c'est l'analyse LR(k)

(LR signifie « **L**eft to right scanning, **R**ightmost derivation »)

En pratique $k = 1$

i.e. on examine uniquement le premier caractère de l'entrée

La pile est de la forme

$$s_0 x_1 s_1 \dots x_n s_n$$

où s_i est un état de l'automate et $x_i \in T \cup N$ comme auparavant

Soit a le premier caractère de l'entrée ; on regarde la transition de l'automate pour l'état s_n et l'entrée a

- si c'est un succès ou un échec, on s'arrête
- si c'est une lecture, alors on empile a et l'état résultat de la transition
- si c'est une réduction $X \rightarrow \alpha$, avec α de longueur p , alors on doit trouver α en sommet de pile

$$s_0 x_1 s_1 \dots x_{n-p} s_{n-p} \mid \alpha_1 s_{n-p+1} \dots \alpha_p s_n$$

on dépile alors α et on empile $X s$, où s est l'état résultat de la transition $s_{n-p} \xrightarrow{X} s$, i.e.

$$s_0 x_1 s_1 \dots x_{n-p} s_{n-p} X s$$

En pratique, on ne travaille pas directement sur l'automate mais sur deux tables

- une table d'**actions** ayant pour lignes les états et pour colonnes les terminaux; la case **action(s, a)** indique
 - shift s' pour une lecture et un nouvel état s'
 - reduce $X \rightarrow \alpha$ pour une réduction
 - un succès
 - un échec
- une table de **déplacements** ayant pour lignes les états et pour colonnes les non terminaux; la case **goto(s, X)** indique l'état résultat d'une réduction de X

Commençons par $k = 0$

On construit un automate LR(0) non déterministe

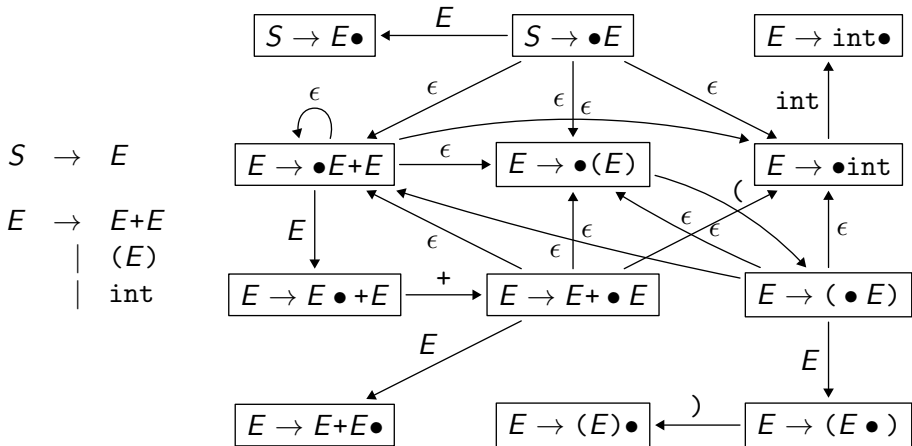
dont les **états** sont des *items* de la forme

$$[X \rightarrow \alpha \bullet \beta]$$

où $X \rightarrow \alpha\beta$ est une production de la grammaire ; l'intuition est : « je cherche à reconnaître X , j'ai déjà lu α et je dois encore lire β »

et les ϵ -**transitions** sont étiquetées par $T \cup N$ et sont les suivantes

$$\begin{array}{ll} [Y \rightarrow \alpha \bullet a\beta] & \xrightarrow{a} [Y \rightarrow \alpha a \bullet \beta] \\ [Y \rightarrow \alpha \bullet X\beta] & \xrightarrow{X} [Y \rightarrow \alpha X \bullet \beta] \\ [Y \rightarrow \alpha \bullet X\beta] & \xrightarrow{\epsilon} [X \rightarrow \bullet \gamma] \quad \text{pour toute production } X \rightarrow \gamma \end{array}$$



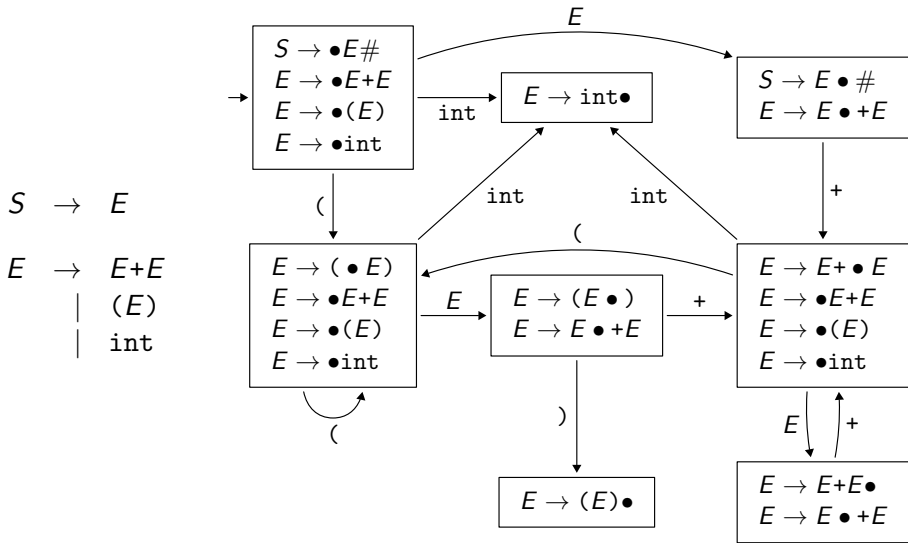
Déterminisons l'automate LR(0)

Pour cela, on regroupe les états reliés par des ϵ -transitions

Les états de l'automate déterministe sont donc des ensembles d'*items*, tel que

$$\begin{aligned} E &\rightarrow E+ \bullet E \\ E &\rightarrow \bullet E+E \\ E &\rightarrow \bullet (E) \\ E &\rightarrow \bullet \text{int} \end{aligned}$$

L'état initial est celui contenant $S \rightarrow \bullet E$



On construit ainsi la table action

- $\text{action}(s, \#) = \text{succès}$ si $[S \rightarrow E \bullet \#] \in s$
- $\text{action}(s, a) = \text{shift } s'$ si on a une transition $s \xrightarrow{a} s'$
- $\text{action}(s, a) = \text{reduce } X \rightarrow \beta$ si $[X \rightarrow \beta \bullet] \in s$, pour tout a
- échec dans tous les autres cas

On construit ainsi la table goto

- $\text{goto}(s, X) = s'$ si et seulement si on a une transition $s \xrightarrow{X} s'$

La table LR(0) peut contenir deux sortes de conflits

- un conflit **lecture/réduction** (*shift/reduce*), si dans un état s on peut effectuer une lecture mais aussi une réduction
- un conflit **réduction/réduction** (*reduce/reduce*), si dans un état s deux réductions différentes sont possibles

Définition (classe LR(0))

Une grammaire est dite LR(0) si la table ainsi construite ne contient pas de conflit.

Sur notre exemple, la table est la suivante :

	action					goto
état	()	+	int	#	E
1	shift 4			shift 2		3
2	reduce $E \rightarrow \text{int}$					
3			shift 6		succès	
4	shift 4			shift 2		5
5		shift 7	shift 6			
6	shift 4			shift 2		8
7	reduce $E \rightarrow (E)$					
8			shift 6			
	reduce $E \rightarrow E+E$					

On a un conflit lecture/réduction dans l'état 8

$$\begin{array}{l} E \rightarrow E + E \bullet \\ E \rightarrow E \bullet + E \end{array}$$

Il illustre précisément l'ambiguïté de la grammaire sur un mot tel que `int+int+int`

On peut résoudre le conflit de deux façons

- si on favorise la **lecture**, on traduira une associativité à droite
- si on favorise la **réduction**, on traduira une associativité à gauche

Privilégions la réduction et illustrons sur un exemple

	()	+	int	#	E
1	s4			s2		3
2	reduce $E \rightarrow \text{int}$					
3			s6		ok	
4	s4			s2		5
5		s7	s6			
6	s4			s2		8
7	reduce $E \rightarrow (E)$					
8	reduce $E \rightarrow E+E$					

pile	entrée	action
1	int+int+int	s2
1 int 2	+int+int	$E \rightarrow \text{int}, g3$
1 E 3	+int+int	s6
1 E 3 + 6	int+int	s2
1 E 3 + 6 int 2	+int	$E \rightarrow \text{int}, g8$
1 E 3 + 6 E 8	+int	$E \rightarrow E+E, g3$
1 E 3	+int	s6
1 E 3 + 6	int	s2
1 E 3 + 6 int 2	#	$E \rightarrow \text{int}, g8$
1 E 3 + 6 E 8	#	$E \rightarrow E+E, g3$
1 E 3	#	succès

La construction LR(0) engendre très facilement des conflits
on va donc chercher à limiter les réductions

Une idée très simple consiste à poser $\text{action}(s, a) = \text{reduce } X \rightarrow \beta$ si et seulement si

$$[X \rightarrow \beta \bullet] \in s \quad \text{et} \quad a \in \text{FOLLOW}(X)$$

Définition (classe SLR(1))

Une grammaire est dite SLR(1) si la table ainsi construite ne contient pas de conflit.

(SLR signifie *Simple LR*)

La grammaire

$$\begin{array}{lcl} S & \rightarrow & E\# \\ E & \rightarrow & E + T \\ & | & T \\ T & \rightarrow & T * F \\ & | & F \\ F & \rightarrow & (E) \\ & | & \text{int} \end{array}$$

est SLR(1)

Exercice : le vérifier (l'automate contient 12 états)

Définition (NULL)

Soit $\alpha \in (T \cup N)^*$. $\text{NULL}(\alpha)$ est vrai si et seulement si on peut dériver ϵ à partir de α i.e. $\alpha \rightarrow^* \epsilon$.

Définition (FIRST)

Soit $\alpha \in (T \cup N)^*$. $\text{FIRST}(\alpha)$ est l'ensemble de tous les premiers terminaux des mots dérivés de α , i.e. $\{a \in T \mid \exists w. \alpha \rightarrow^* aw\}$.

Définition (FOLLOW)

Soit $X \in N$. $\text{FOLLOW}(X)$ est l'ensemble de tous les terminaux qui peuvent apparaître après X dans une dérivation, i.e. $\{a \in T \mid \exists u, w. S \rightarrow^* uXaw\}$.

Pour calculer $\text{NULL}(\alpha)$ il suffit de déterminer $\text{NULL}(X)$ pour $X \in N$

$\text{NULL}(X)$ est vrai si et seulement si

- il existe une production $X \rightarrow \epsilon$,
- ou il existe une production $X \rightarrow Y_1 \dots Y_m$ où $\text{NULL}(Y_i)$ pour tout i

Problème : il s'agit donc d'un ensemble d'équations mutuellement récursives

Dit autrement, si $N = \{X_1, \dots, X_n\}$ et si $\vec{V} = (\text{NULL}(X_1), \dots, \text{NULL}(X_n))$, on cherche la **plus petite solution** d'une équation de la forme

$$\vec{V} = F(\vec{V})$$

Théorème (existence d'un plus petit point fixe (Tarski))

Soit A un ensemble fini muni d'une relation d'ordre \leq et d'un plus petit élément ε . Toute fonction $f : A \rightarrow A$ croissante, i.e. telle que $\forall x, y. x \leq y \Rightarrow f(x) \leq f(y)$, admet un plus petit point fixe.

Preuve : comme ε est le plus petit élément, on a $\varepsilon \leq f(\varepsilon)$
 f étant croissante, on a donc $f^k(\varepsilon) \leq f^{k+1}(\varepsilon)$ pour tout k
 A étant fini, il existe donc un plus petit k_0 tel que $f^{k_0}(\varepsilon) = f^{k_0+1}(\varepsilon)$
 $a_0 = f^{k_0}(\varepsilon)$ est donc un point fixe de f

soit b un autre point fixe de f
on a $\varepsilon \leq b$ et donc $f^k(\varepsilon) \leq f^k(b)$ pour tout k
en particulier $a_0 = f^{k_0}(\varepsilon) \leq f^{k_0}(b) = b$
 a_0 est donc le plus petit point fixe de f



Dans le cas du calcul de NULL, on a $A = \text{BOOL} \times \dots \times \text{BOOL}$ avec $\text{BOOL} = \{\text{false}, \text{true}\}$

On peut munir BOOL de l'ordre $\text{false} \leq \text{true}$ et A de l'ordre point à point

$$(x_1, \dots, x_n) \leq (y_1, \dots, y_n) \quad \text{si et seulement si} \quad \forall i. x_i \leq y_i$$

Le théorème s'applique alors en prenant

$$\varepsilon = (\text{false}, \dots, \text{false})$$

car la fonction calculant $\text{NULL}(X)$ à partir des $\text{NULL}(X_i)$ est croissante

Pour calculer les $\text{NULL}(X_i)$, on part donc de

$$\text{NULL}(X_1) = \text{false}, \dots, \text{NULL}(X_n) = \text{false}$$

et on applique les équations jusqu'à obtention du point fixe *i.e.* jusqu'à ce que la valeur des $\text{NULL}(X_i)$ ne soit plus modifiée

$$\begin{aligned}
 E &\rightarrow T E' \\
 E' &\rightarrow + T E' \\
 &\quad | \quad \epsilon \\
 T &\rightarrow F T' \\
 T' &\rightarrow * F T' \\
 &\quad | \quad \epsilon \\
 F &\rightarrow (E) \\
 &\quad | \quad \text{int}
 \end{aligned}$$

E	E'	T	T'	F
false	false	false	false	false
false	true	false	true	false
false	true	false	true	false

Pourquoi cherche-t-on le **plus petit** point fixe ?

- ⇒ par récurrence sur le nombre d'étapes du calcul précédent, on montre que si $\text{NULL}(X) = \text{true}$ alors $X \rightarrow^* \epsilon$
- ⇐ par récurrence sur le nombre d'étapes de la dérivation $X \rightarrow^* \epsilon$, on montre que $\text{NULL}(X) = \text{true}$ par le calcul précédent

De même, les équations définissant FIRST sont mutuellement récursives

$$\text{FIRST}(X) = \bigcup_{X \rightarrow \beta} \text{FIRST}(\beta)$$

et

$$\begin{aligned} \text{FIRST}(\epsilon) &= \emptyset \\ \text{FIRST}(a\beta) &= \{a\} \\ \text{FIRST}(X\beta) &= \text{FIRST}(X), \quad \text{si } \neg \text{NULL}(X) \\ \text{FIRST}(X\beta) &= \text{FIRST}(X) \cup \text{FIRST}(\beta), \quad \text{si } \text{NULL}(X) \end{aligned}$$

De même, on procède par calcul de point fixe sur le produit cartésien $A = \mathcal{P}(T) \times \cdots \times \mathcal{P}(T)$ muni, point à point, de l'ordre \subseteq et avec $\varepsilon = (\emptyset, \dots, \emptyset)$

$$\begin{aligned}
 E &\rightarrow T E' \\
 E' &\rightarrow + T E' \\
 &\quad | \quad \epsilon \\
 T &\rightarrow F T' \\
 T' &\rightarrow * F T' \\
 &\quad | \quad \epsilon \\
 F &\rightarrow (E) \\
 &\quad | \quad \text{int}
 \end{aligned}$$

NULL

E	E'	T	T'	F
false	true	false	true	false

FIRST

E	E'	T	T'	F
\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
\emptyset	$\{+\}$	\emptyset	$\{*\}$	$\{(, \text{int})\}$
\emptyset	$\{+\}$	$\{(, \text{int})\}$	$\{*\}$	$\{(, \text{int})\}$
$\{(, \text{int})\}$	$\{+\}$	$\{(, \text{int})\}$	$\{*\}$	$\{(, \text{int})\}$
$\{(, \text{int})\}$	$\{+\}$	$\{(, \text{int})\}$	$\{*\}$	$\{(, \text{int})\}$

Là encore, les équations définissant FOLLOW sont mutuellement récursives

$$\text{FOLLOW}(X) = \bigcup_{Y \rightarrow \alpha X \beta} \text{FIRST}(\beta) \cup \bigcup_{Y \rightarrow \alpha X \beta, \text{NULL}(\beta)} \text{FOLLOW}(Y)$$

On procède par calcul de point fixe, sur le même domaine que pour FIRST

Note : il faut introduire $\#$ dans les suivants du symbole de départ (ce que l'on peut faire directement, ou en ajoutant une règle $S' \rightarrow S\#$)

NULL

E	E'	T	T'	F
false	true	false	true	false

$$\begin{aligned}
 E &\rightarrow T E' \\
 E' &\rightarrow + T E' \\
 &\quad | \epsilon \\
 T &\rightarrow F T' \\
 T' &\rightarrow * F T' \\
 &\quad | \epsilon \\
 F &\rightarrow (E) \\
 &\quad | \text{int}
 \end{aligned}$$

FIRST

E	E'	T	T'	F
$\{ (, \text{int} \}$	$\{ + \}$	$\{ (, \text{int} \}$	$\{ * \}$	$\{ (, \text{int} \}$

FOLLOW

E	E'	T	T'	F
$\{ \# \}$	\emptyset	\emptyset	\emptyset	\emptyset
$\{ \#,) \}$	$\{ \# \}$	$\{ +, \# \}$	\emptyset	$\{ * \}$
$\{ \#,) \}$	$\{ \#,) \}$	$\{ +, \#,) \}$	$\{ +, \# \}$	$\{ *, +, \# \}$
$\{ \#,) \}$	$\{ \#,) \}$	$\{ +, \#,) \}$	$\{ +, \#,) \}$	$\{ *, +, \#,) \}$
$\{ \#,) \}$	$\{ \#,) \}$	$\{ +, \#,) \}$	$\{ +, \#,) \}$	$\{ *, +, \#,) \}$

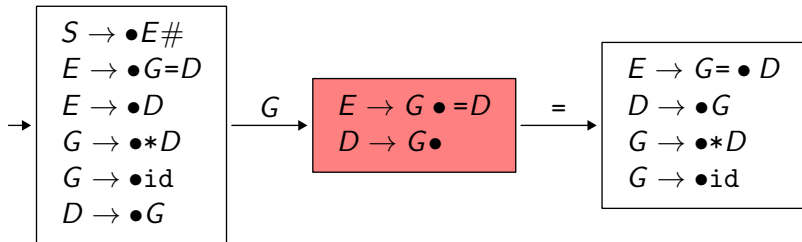
Limites de l'analyse SLR(1)

En pratique, la classe SLR(1) reste trop restrictive

exemple :

$$\begin{aligned} S &\rightarrow E\# \\ E &\rightarrow G = D \\ &\quad | D \\ G &\rightarrow * D \\ &\quad | \text{id} \\ D &\rightarrow G \end{aligned}$$

	=	
1
2	shift 3 reduce $D \rightarrow G$...
3	\vdots	\ddots



On introduit une classe de grammaires encore plus large, **LR(1)**, au prix de tables encore plus grandes

Dans l'analyse LR(1), les *items* ont maintenant la forme

$$[X \rightarrow \alpha \bullet \beta, a]$$

dont la signification est : « je cherche à reconnaître X , j'ai déjà lu α et je dois encore lire β puis vérifier que le caractère suivant est a »

Les transitions de l'automate LR(1) non déterministe sont

$$\begin{aligned}
 [Y \rightarrow \alpha \bullet a\beta, b] &\xrightarrow{a} [Y \rightarrow \alpha a \bullet \beta, b] \\
 [Y \rightarrow \alpha \bullet X\beta, b] &\xrightarrow{X} [Y \rightarrow \alpha X \bullet \beta, b] \\
 [Y \rightarrow \alpha \bullet X\beta, b] &\xrightarrow{\epsilon} [X \rightarrow \bullet \gamma, c] \quad \text{pour tout } c \in \text{FIRST}(\beta b)
 \end{aligned}$$

L'état initial est celui qui contient $[S \rightarrow \bullet \alpha, \#]$

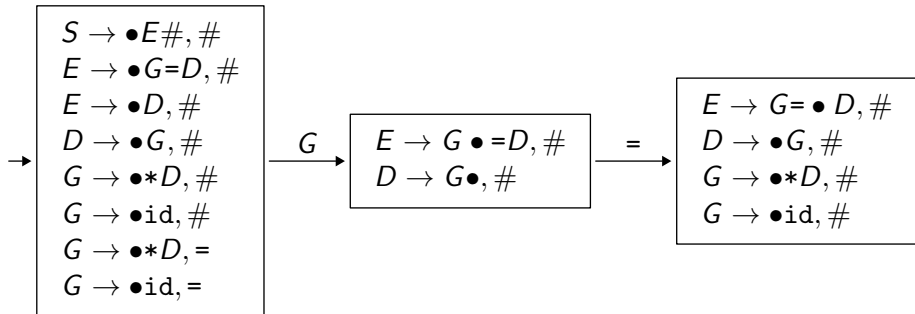
Comme précédemment, on peut déterminer l'automate et construire la table correspondante ; on introduit une action de réduction pour (s, a) seulement lorsque s contient un item de la forme $[X \rightarrow \alpha \bullet, a]$

Définition (classe LR(1))

Une grammaire est dite LR(1) si la table ainsi construite ne contient pas de conflit.

$S \rightarrow E\#$
 $E \rightarrow G = D$
 $\quad \mid D$
 $G \rightarrow * D$
 $\quad \mid \text{id}$
 $D \rightarrow G$

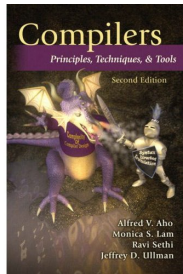
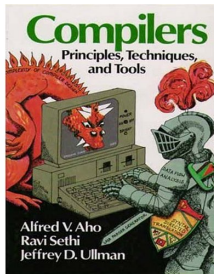
	#	=	
1
2	reduce $D \rightarrow G$	shift 3	...
3	:	:	..



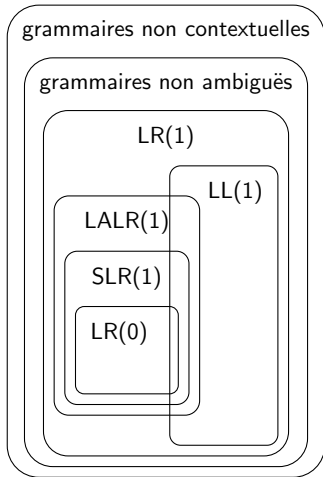
La construction LR(1) pouvant être coûteuse, il existe des approximations

La classe LALR(1) (*lookahead LR*) est une telle approximation, utilisée notamment dans les outils de la famille yacc

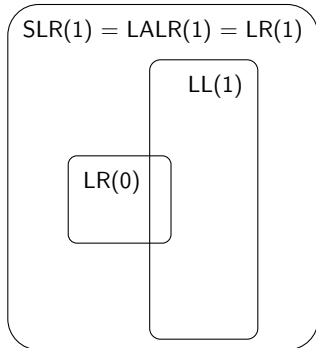
Plus d'info : voir par exemple *Compilateurs : principes techniques et outils* (dit « le dragon ») de A. Aho, R. Sethi, J. Ullman, section 4.7



grammaires



langages



L'analyse ascendante est puissante mais le calcul des tables est complexe

Le travail est automatisé par de nombreux outils

C'est la grande famille de yacc, bison, ocaml yacc, cups, menhir, ...
(YACC signifie *Yet Another Compiler Compiler*)

l'outil Menhir

Menhir est un outil qui transforme une grammaire en un analyseur OCaml ; Menhir est basé sur une analyse LR(1)

Chaque production de la grammaire est accompagnée d'une **action sémantique** *i.e.* du code OCaml construisant une valeur sémantique (typiquement un arbre de syntaxe abstraite)

Menhir s'utilise conjointement avec un analyseur lexical (tel `ocamllex`)

Un fichier Menhir porte le suffixe `.mly` et a la structure suivante

```
%{  
    ... code OCaml arbitraire ...  
%}  
...déclaration des tokens...  
...déclaration des précédences et associativités...  
...déclaration des points d'entrée...  
%%  
non-terminal-1:  
| production { action }  
| production { action }  
;  
  
non-terminal-2:  
| production { action }  
...  
%%  
    ... code OCaml arbitraire ...
```

```
%token PLUS LPAR RPAR EOF
```

```
%token <int> INT
```

```
%start <int> phrase
```

```
%%
```

```
phrase:
```

```
    e = expression; EOF  { e }
```

```
;
```

```
expression:
```

```
| e1 = expression; PLUS; e2 = expression { e1 + e2 }
```

```
| LPAR; e = expression; RPAR              { e }
```

```
| i = INT                                { i }
```

```
;
```

On compile le fichier `arith.mly` de la manière suivante

```
% menhir -v arith.mly
```

On obtient du code OCaml pur dans `arith.ml(i)`, qui contient notamment

La déclaration d'un type `token`

```
type token = RPAR | PLUS | LPAR | INT of int | EOF
```

et, pour chaque non terminal déclaré avec `%start`, une fonction du type

```
val phrase: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int
```

Comme on le voit, cette fonction prend en argument un analyseur lexical, du type de celui produit par `ocamllex` (cf. cours précédent)

Lorsque la grammaire n'est pas LR(1), Menhir présente les **conflits** à l'utilisateur

- le fichier `.automaton` contient une description de l'automate LR(1); les conflits y sont mentionnés
- le fichier `.conflicts` contient, le cas échéant, une explication de chaque conflit, sous la forme d'une séquence de lexèmes conduisant à deux arbres de dérivation

Sur la grammaire ci-dessus, Menhir signale un conflit

```
% menhir -v arith.mly  
Warning: one state has shift/reduce conflicts.  
Warning: one shift/reduce conflict was arbitrarily resolved.
```

Le fichier `arith.automaton` contient notamment

```
State 6:  
expression -> expression . PLUS expression [ RPAR PLUS EOF ]  
expression -> expression PLUS expression . [ RPAR PLUS EOF ]  
-- On PLUS shift to state 5  
-- On RPAR reduce production expression -> expression PLUS expression  
-- On PLUS reduce production expression -> expression PLUS expression  
-- On EOF reduce production expression -> expression PLUS expression  
** Conflict on PLUS
```

Le fichier arith.conflicts contient une explication limpide

```
** Conflict (shift/reduce) in state 6.
```

```
** Token involved: PLUS
```

```
** This state is reached from phrase after reading:
```

```
expression PLUS expression
```

```
** In state 6, looking ahead at PLUS, shifting is permitted
```

```
** because of the following sub-derivation:
```

```
expression PLUS expression
```

```
        expression . PLUS expression
```

```
** In state 6, looking ahead at PLUS, reducing production
```

```
** expression -> expression PLUS expression
```

```
** is permitted because of the following sub-derivation:
```

```
expression PLUS expression // lookahead token appears
```

Une manière de résoudre les conflits est d'indiquer à Menhir comment choisir entre lecture et réduction

Pour cela, on donne des **priorités** aux lexèmes et aux productions, et des règles d'**associativité**

Par défaut, la priorité d'une production est celle de son lexème le plus à droite (mais elle peut être spécifiée explicitement)

Les priorités des lexèmes sont données explicitement par l'utilisateur

Si la priorité de la production est supérieure à celle du lexème à lire, alors la réduction est favorisée

Inversement, si la priorité du lexème est supérieure, alors la lecture est favorisée

En cas d'égalité, l'associativité est consultée : un lexème associatif à gauche favorise la réduction, et un lexème associatif à droite la lecture

Dans notre exemple, il suffit d'indiquer par exemple que PLUS est associatif à gauche

```
%token PLUS LPAR RPAR EOF
%token <int> INT
%left PLUS
%start <int> phrase
%%
phrase:
    e = expression; EOF { e }
;
expression:
| e1 = expression; PLUS; e2 = expression { e1 + e2 }
| LPAR; e = expression; RPAR { e }
| i = INT { i }
;
```

Pour associer des priorités aux lexèmes, on utilise la convention suivante :

- l'ordre de déclaration des associativités fixe les priorités (les premiers lexèmes ont les priorités les plus faibles)
- plusieurs lexèmes peuvent apparaître sur la même ligne, ayant ainsi la même priorité

Exemple :

```
%left PLUS MINUS  
%left TIMES DIV
```

Des règles très ressemblantes...

```
%token PLUS MINUS MULT DIV
```

```
%left PLUS MINUS
```

```
%left MULT DIV
```

```
%start <int> expression
```

```
%%
```

```
expression:
```

```
| e1 = expression; PLUS; e2 = expression { e1 + e2 }
```

```
| e1 = expression; MINUS; e2 = expression { e1 - e2 }
```

```
| e1 = expression; MULT; e2 = expression { e1 * e2 }
```

```
| e1 = expression; DIV; e2 = expression { e1 / e2 }
```

```
;
```

... peuvent être résumées

```
%token PLUS MINUS MULT DIV
%left PLUS MINUS
%left MULT DIV
%start <int> expression
%%
expression:
| e1 = expressoin; op = binop; e2 = expression { op e1 e2 }
;
binop:
| PLUS    { ( + ) }
| MINUS   { ( - ) }
| MULT    { ( * ) }
| DIV     { ( / ) }
;
```

Mais on y perd les priorités !

Pour garder les priorités, on peut écrire la version courte et demander à Menhir de transformer en la version longue :

```
%token PLUS MINUS MULT DIV
%left PLUS MINUS
%left MULT DIV
%start <int> expression
%%
expression:
| e1 = expressoin; op = binop; e2 = expression { op e1 e2 }
;
%inline binop:
| PLUS    { ( + ) }
| MINUS   { ( - ) }
| MULT    { ( * ) }
| DIV     { ( / ) }
;
```

Menhir offre de nombreux avantages par rapport aux outils traditionnels tels que `ocaml yacc` :

- non-terminaux paramétrés par des (non-)terminaux
 - en particulier, facilités pour écrire des expressions régulières ($E?$, E^* , E^+), des listes avec séparateur
- explication des conflits
- mode interactif
- analyse LR(1) plutôt que LALR(1)

Pour que les phases suivantes de l'analyse (typiquement le typage) puissent **localiser** les messages d'erreur, il convient de conserver une information de localisation dans l'arbre de syntaxe abstraite

Menhir fournit cette information dans `$startpos` et `$endpos`, deux valeurs du type `Lexing.position`; cette information lui a été transmise par l'analyseur lexical

Attention : `ocamllex` ne maintient automatiquement que la position absolue dans le fichier; pour avoir les numéros de ligne et de colonnes à jour, il faut un traitement spécial du retour chariot dans l'analyseur lexical (voir par exemple `lexer.mll` fourni dans le TP 2)

Une façon de conserver l'information de localisation dans l'arbre de syntaxe abstraite est la suivante (cf cours 4)

```
type expression =  
  { desc: desc;  
    loc : Lexing.position * Lexing.position }  
  
and desc =  
  | Econst of int  
  | Eplus  of expression * expression  
  | Eneg   of expression  
  | ...
```


La grammaire peut donc ressembler à ceci

expression:

```
| d = desc { { desc = d; loc = $startpos, $endpos } }  
;
```

desc:

```
| i = INT { Econst i }  
| e1 = expression; PLUS; e2 = expression { Eplus (e1, e2) }  
| ...
```