École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

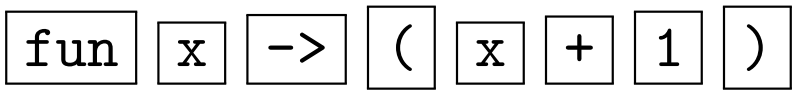
analyse syntaxique (1/2)

quel point commun ?

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

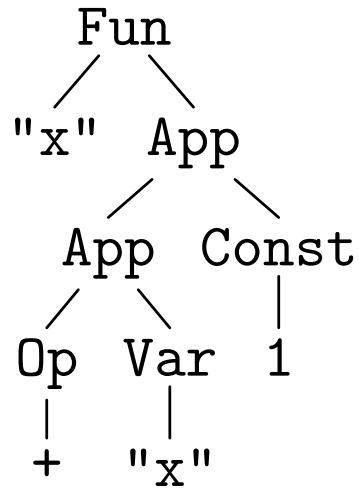
son entrée est le flot des lexèmes construits par l'analyse lexicale,
sa sortie est un arbre de syntaxe abstraite

suite de lexèmes



analyse syntaxique

syntaxe abstraite



erreurs de syntaxe

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

plan

1. grammaires
2. analyse ascendante
3. l'outil Menhir
4. derrière l'outil Menhir

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***)*
- $R \subseteq N \times (N \cup T)^\star$ *est un ensemble fini de* **règles de production**

exemple : expressions arithmétiques

$N = \{E\}$, $T = \{+, *, (,), \texttt{int}\}$, $S = E$,
et $R = \{ (E, E+E), (E, E*E), (E, (E)), (E, \texttt{int}) \}$

en pratique, on note les règles sous la forme

$$\begin{array}{rcl} E & \rightarrow & E + E \\ & | & E * E \\ & | & (E) \\ & | & \texttt{int} \end{array}$$

les terminaux de la grammaire seront les lexèmes produits par l'analyse lexicale

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

Définition

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

$$u = u_1 X u_2$$

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

$$v = u_1 \beta u_2$$

exemple :

$$\underbrace{E * (}_{u_1} \; \underbrace{E}_{X} \; \underbrace{)}_{u_2} \quad \to \quad E * (\; \underbrace{E + E}_{\beta} \;)$$

une suite $w_1 \to w_2 \to \cdots \to w_n$ est appelée une dérivation

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

on note $\to^\star$ la clôture réflexive transitive de $\to$

exemple de dérivation gauche :

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

on a donc (en particulier, mais pas uniquement)

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

Définition

Le **langage** *défini par une grammaire non contextuelle* $G = (N, T, S, R)$ *est l'ensemble des mots de* $T^\star$ *dérivés de l'axiome, i.e.*

$$L(G) = \{ w \in T^\star \mid S \rightarrow^\star w \}$$

dans notre exemple

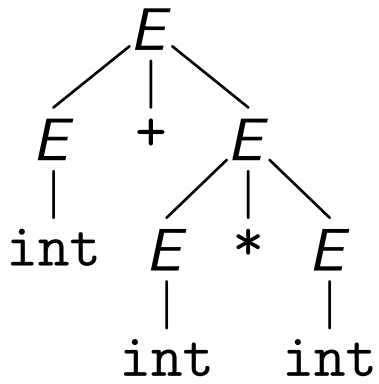
$$\texttt{int * (int + int)} \in L(G)$$

Définition

Un **arbre de dérivation** *est un arbre dont les nœuds sont étiquetés par des symboles de la grammaire, de la manière suivante :*

- *la racine est l'axiome S ;*
- *tout nœud interne X est un non terminal dont les fils sont étiquetés par $\beta \in (N \cup T)^\star$ avec $X \to \beta$ une règle de la dérivation*

exemple :



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

pour un arbre de dérivation dont les feuilles forment le mot w dans l'ordre infixe, il est clair qu'on a $S \rightarrow^\star w$

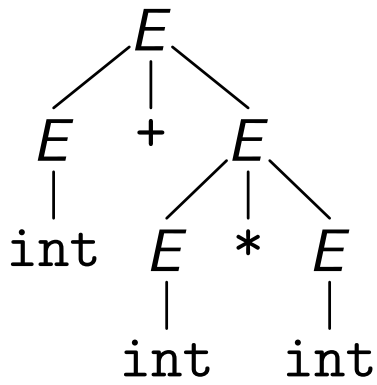
inversement, à toute dérivation $S \rightarrow^\star w$, on peut associer un arbre de dérivation dont les feuilles forment le mot w dans l'ordre infixe

idée : l'arbre de dérivation capture tout un ensemble de dérivations que l'on souhaite identifier

la dérivation gauche

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

donne l'arbre de dérivation



mais la dérivation droite

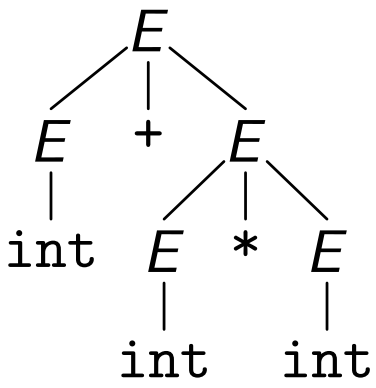
$$E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * \texttt{int} \rightarrow E + \texttt{int} * \texttt{int}$$
$$\rightarrow \texttt{int} + \texttt{int} * \texttt{int}$$

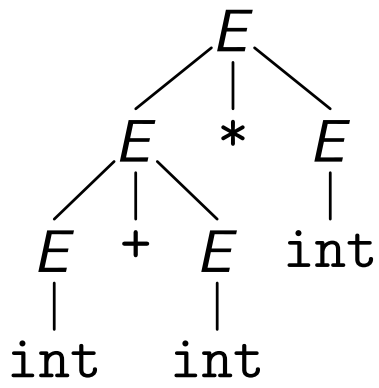
également

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



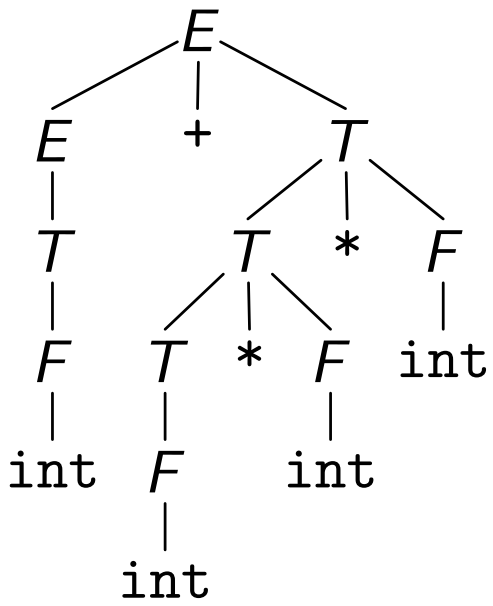


pour ce langage-là, il est néanmoins possible de proposer une autre grammaire, non ambiguë, qui définit le même langage

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

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

$$
\begin{aligned}
E &\to E + T \to T + T \to F + T \to \texttt{int} + T \to \texttt{int} + T * F \\
&\to \texttt{int} + T * F * F \to \texttt{int} + F * F * F \\
&\to \texttt{int} + \texttt{int} * F * F \to \texttt{int} + \texttt{int} * \texttt{int} * F \\
&\to \texttt{int} + \texttt{int} * \texttt{int} * \texttt{int}
\end{aligned}
$$

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

(rappel : décidable veut dire qu'on peut écrire un programme qui, pour toute entrée, termine et répond oui ou non)

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 LR(0), SLR(1), LALR(1), LR(1), LL(1), etc.

analyse ascendante

idée générale

- lire l'entrée de gauche à droite
- 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 $(T \cup N)^\star$

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 $X \to \beta$, 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

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

pile	entrée	action
ϵ	`int+int*int`	lecture
`int`	`+int*int`	réduction $F \rightarrow$ `int`
F	`+int*int`	réduction $T \rightarrow F$
T	`+int*int`	réduction $E \rightarrow T$
E	`+int*int`	lecture
E`+`	`int*int`	lecture
E`+int`	`*int`	réduction $F \rightarrow$ `int`
$E+F$	`*int`	réduction $T \rightarrow F$
$E+T$	`*int`	lecture
$E+T$`*`	`int`	lecture
$E+T*$`int`		réduction $F \rightarrow$ `int`
$E+T*F$		réduction $T \rightarrow T*F$
$E+T$		réduction $E \rightarrow E+T$
E		succès

analyse LR (Knuth, 1965)

comment prendre la décision lecture / réduction ?

en se servant d'un automate fini et en examinant les k premiers lexèmes 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 lexème de l'entrée

la pile est de la forme

$$s_0\, x_1\, s_1\, \ldots\, x_n\, s_n$$

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

soit a le premier lexème de l'entrée
une table indexée par s_n et a nous indique l'action à effectuer

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

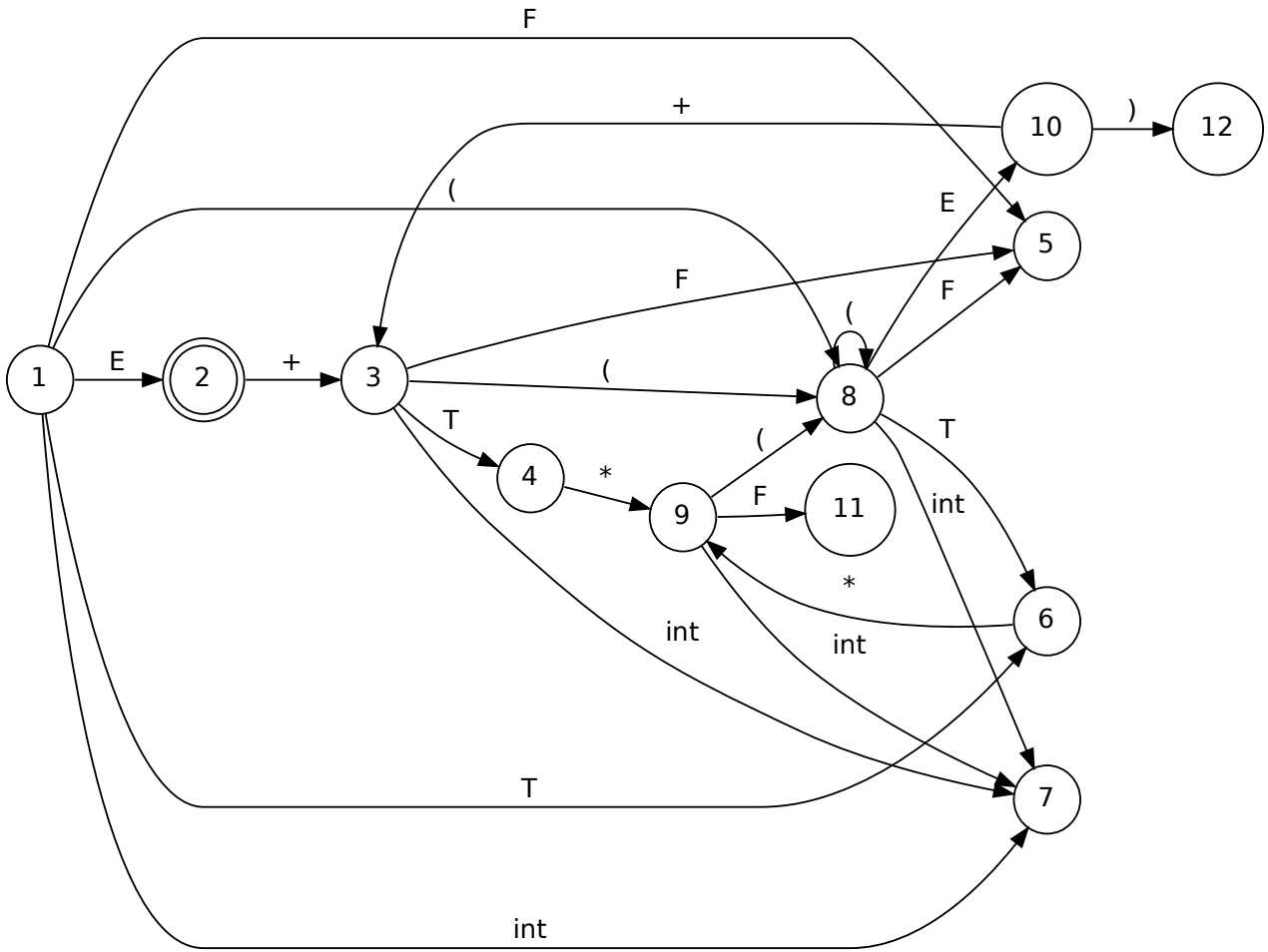
$$s_0\, x_1\, s_1\, \ldots\, x_{n-p}\, s_{n-p}\, |\alpha_1\, s_{n-p+1} \ldots\, \alpha_p\, s_n$$

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

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

dans l'exemple plus haut, on s'est servi de cet automate

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



en pratique, on ne travaille pas avec l'automate mais avec deux tables

- une table d'**actions** ayant pour lignes les états et pour colonnes les terminaux ; la case $\texttt{action}(s, a)$ indique
  - `shift` s' pour une lecture et un nouvel état s'
  - `reduce` $X \to \alpha$ pour une réduction
  - un succès
  - un échec

- une table de **déplacements** ayant pour lignes les états et pour colonnes les non terminaux ; la case $\texttt{goto}(s, X)$ indique l'état résultat d'une réduction de X

fin de l'entrée

on ajoute aussi un lexème spécial, noté #, qui désigne la fin de l'entrée

on peut le voir comme l'ajout d'un nouveau non terminal S (qui devient l'axiome) et d'une nouvelle règle

$$\begin{array}{lcl} S & \rightarrow & E\ \# \\ E & \rightarrow & \ldots \\ \vdots & & \end{array}$$

sur notre exemple, les tables sont les suivantes :

1	E	$\rightarrow$	E + T
2		$\mid$	T
3	T	$\rightarrow$	T * F
4		$\mid$	F
5	F	$\rightarrow$	(E)
6		$\mid$	`int`

si = `shift` i

rj = `reduce` j

	action						*goto*		
état	(	)	+	*	`int`	#	E	T	F
1	$s8$				$s7$		2	6	5
2			$s3$			succès			
3	$s8$				$s7$			4	5
4		$r1$	$r1$	$s9$		$r1$			
5		$r4$	$r4$	$r4$		$r4$			
6		$r2$	$r2$	$s9$		$r2$			
7		$r6$	$r6$	$r6$		$r6$			
8	$s8$				$s7$		10	6	5
9	$s8$				$s7$				11
10		$s12$	$s3$						
11		$r3$	$r3$	$r3$		$r3$			
12		$r5$	$r5$	$r5$		$r5$			

exemple d'exécution

pile	entrée	action
1	`int+int*int#`	s7
1 `int` 7	`+int*int#`	$F \rightarrow$ `int`, g5
1 F 5	`+int*int#`	$T \rightarrow F$, g6
1 T 6	`+int*int#`	$E \rightarrow T$, g2
1 E 2	`+int*int#`	s3
1 E 2 + 3	`int*int#`	s7
1 E 2 + 3 `int` 7	`*int#`	$F \rightarrow$ `int`, g5
1 E 2 + 3 F 5	`*int#`	$T \rightarrow F$, g4
1 E 2 + 3 T 4	`*int#`	s9
1 E 2 + 3 T 4 * 9	`int#`	s7
1 E 2 + 3 T 4 * 9 `int` 7	#	$F \rightarrow$ `int`, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T*F$, g4
1 E 2 + 3 T 4	#	$E \rightarrow E+T$, g2
1 E 2	#	succès

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`, `ocamlyacc`, `cup`, `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 (typiquement `ocamllex`)

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

```
%{
  ... code OCaml arbitraire ...
%}
...déclaration des lexèmes...
...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
```

- 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 3)

quand on combine `ocamllex` et `menhir`

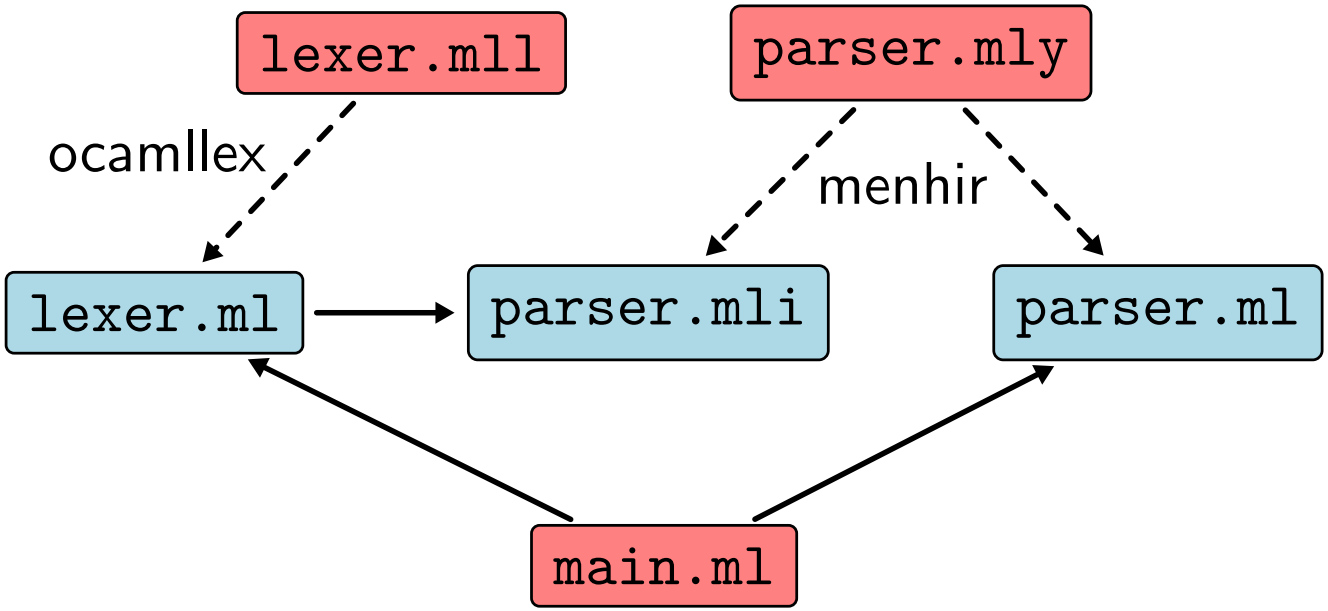
- `lexer.mll` fait référence aux lexèmes définis dans `parser.mly`

```
{
  open Parser
}
...
```

- l'analyseur lexical et l'analyseur syntaxique sont combinés ainsi :

```
let c  = open_in file in
let lb = Lexing.from_channel c in
let e  = Parser.phrase Lexer.token lb in
...
```

ocamllex + menhir



= source utilisateur

= construit automatiquement

= dépendance

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.
```

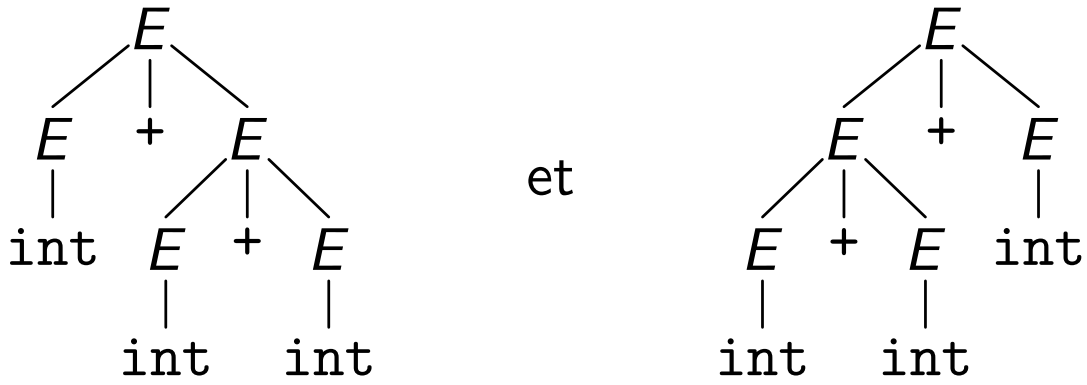
le fichier `arith.automaton` contient notamment

```
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 expressi
-- On PLUS reduce production expression -> expression PLUS expressi
-- On EOF reduce production expression -> expression PLUS expressio
** 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
```

dit autrement, la question est de choisir entre



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

pour cela, on peut donner 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)

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
```

la grammaire suivante contient un conflit

```
expression:
| IF e1 = expression; THEN; e2 = expression
    { ... }
| IF e1 = expression; THEN; e2 = expression;
  ELSE; e3 = expression
    { ... }
| i = INT
    { ... }
| ...
```

(connu en anglais sous le nom de *dangling else*)

il correspond à la situation

```
IF a THEN IF b THEN c ELSE d
```

pour associer le `ELSE` au `THEN` le plus proche, il faut privilégier la lecture

```
%nonassoc THEN
%nonassoc ELSE
```

atouts de Menhir

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

- non-terminaux paramétrés par des (non-)terminaux
  - en particulier, facilités pour écrire des expressions régulières ($E?$, $E*$, $E+$) et des listes avec séparateur
- explication des conflits
- mode interactif
- analyse LR(1), là où la plupart des outils n'offrent que LALR(1)

lire le manuel de Menhir ! (accessible depuis la page du cours)

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 appeler `Lexing.new_line` pour chaque retour chariot (cf code fourni au TD 2)

une façon de conserver l'information de localisation dans l'arbre de syntaxe abstraite est la suivante

```
type expression =
  { desc: desc;
    loc : Lexing.position * Lexing.position }

and desc =
  | Econst of int
  | Eplus  of expression * expression
  | Eneg   of expression
  | ...
```

chaque nœud est ainsi décoré par une localisation

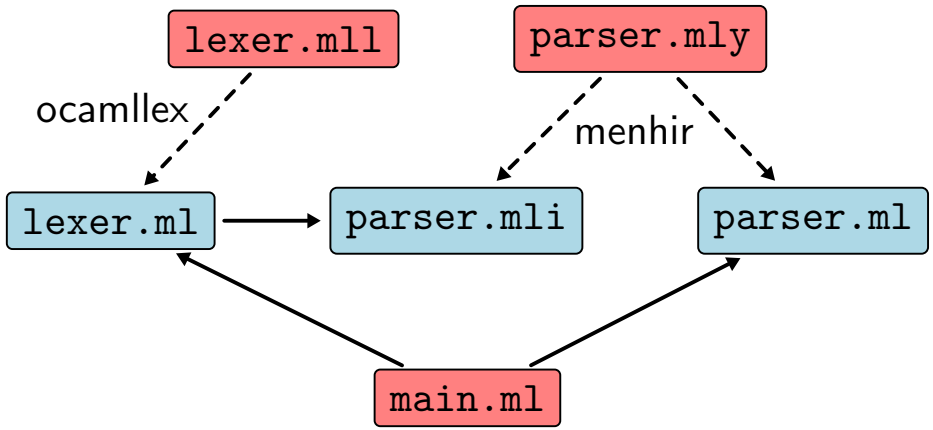
la grammaire peut donc ressembler à ceci

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

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

comme dans le cas d'`ocamllex`, il faut s'assurer de l'application de `menhir` avant le calcul des dépendances

si on utilise `dune`, on indique la présence d'un fichier `menhir` :

```
(ocamllex
 (modules lexer))
(menhir
 (flags --explain --dump)
 (modules parser))
(executable
 (name minilang)
 ...
```



(cf le code fourni avec le TD 4 par exemple)

construction de l'automate et des tables

Définition (NULL)

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

Définition (FIRST)

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

Définition (FOLLOW)

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

calcul de NULL, FIRST et FOLLOW

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

problème : il s'agit 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 $f : A \to A$ croissante, i.e. telle que $\forall x, y \,.\, x \leq y \Rightarrow f(x) \leq f(y)$, admet un plus petit point fixe.

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

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 □

note : ce sont là des conditions suffisantes mais pas nécessaires

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

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

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

le théorème s'applique alors en prenant

$$\varepsilon = (\texttt{false}, \ldots, \texttt{false})$$

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

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

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

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

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & \texttt{+}\ T\ E' \\
 & | & \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & \texttt{*}\ F\ T' \\
 & | & \epsilon \\
F & \rightarrow & \texttt{(}\ E\ \texttt{)} \\
 & | & \texttt{int}
\end{array}
$$

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

$\Rightarrow$ par récurrence sur le nombre d'étapes du calcul précédent, on montre que si $\text{NULL}(X) = \texttt{true}$ alors $X \rightarrow^\star \epsilon$

$\Leftarrow$ par récurrence sur le nombre d'étapes de la dérivation $X \rightarrow^\star \epsilon$, on montre que $\text{NULL}(X) = \texttt{true}$ par le calcul précédent

de même, les équations définissant FIRST sont mutuellement récursives

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

et

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

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 $\varepsilon = (\emptyset, \ldots, \emptyset)$

NULL

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

FIRST

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

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & \texttt{+}\ T\ E' \\
 & | & \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & \texttt{*}\ F\ T' \\
 & | & \epsilon \\
F & \rightarrow & \texttt{(}\ E\ \texttt{)} \\
 & | & \texttt{int}
\end{array}
$$

là encore, les équations définissant FOLLOW sont mutuellement récursives

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

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 $S' \to S\#$)

NULL

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

FIRST

E	E'	T	T'	F
{`(`, `int`}	{`+`}	{`(`, `int`}	{`*`}	{`(`, `int`}

FOLLOW

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

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & \texttt{+}\ T\ E' \\
 & | & \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & \texttt{*}\ F\ T' \\
 & | & \epsilon \\
F & \rightarrow & \texttt{(}\ E\ \texttt{)} \\
 & | & \texttt{int}
\end{array}
$$

calculer NULL, FIRST et FOLLOW pour la grammaire « de LISP »

$$
\begin{array}{rcl}
S & \rightarrow & E\ \# \\
E & \rightarrow & \texttt{sym} \\
 & | & (\ L\ ) \\
L & \rightarrow & \epsilon \\
 & | & E\ L
\end{array}
$$

fixons pour l'instant $k = 0$

on commence par construire un automate **asynchrone**

c'est-à-dire contenant des transitions spontanées
appelées ϵ-**transitions** et notées $s_1 \xrightarrow{\epsilon} s_2$

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

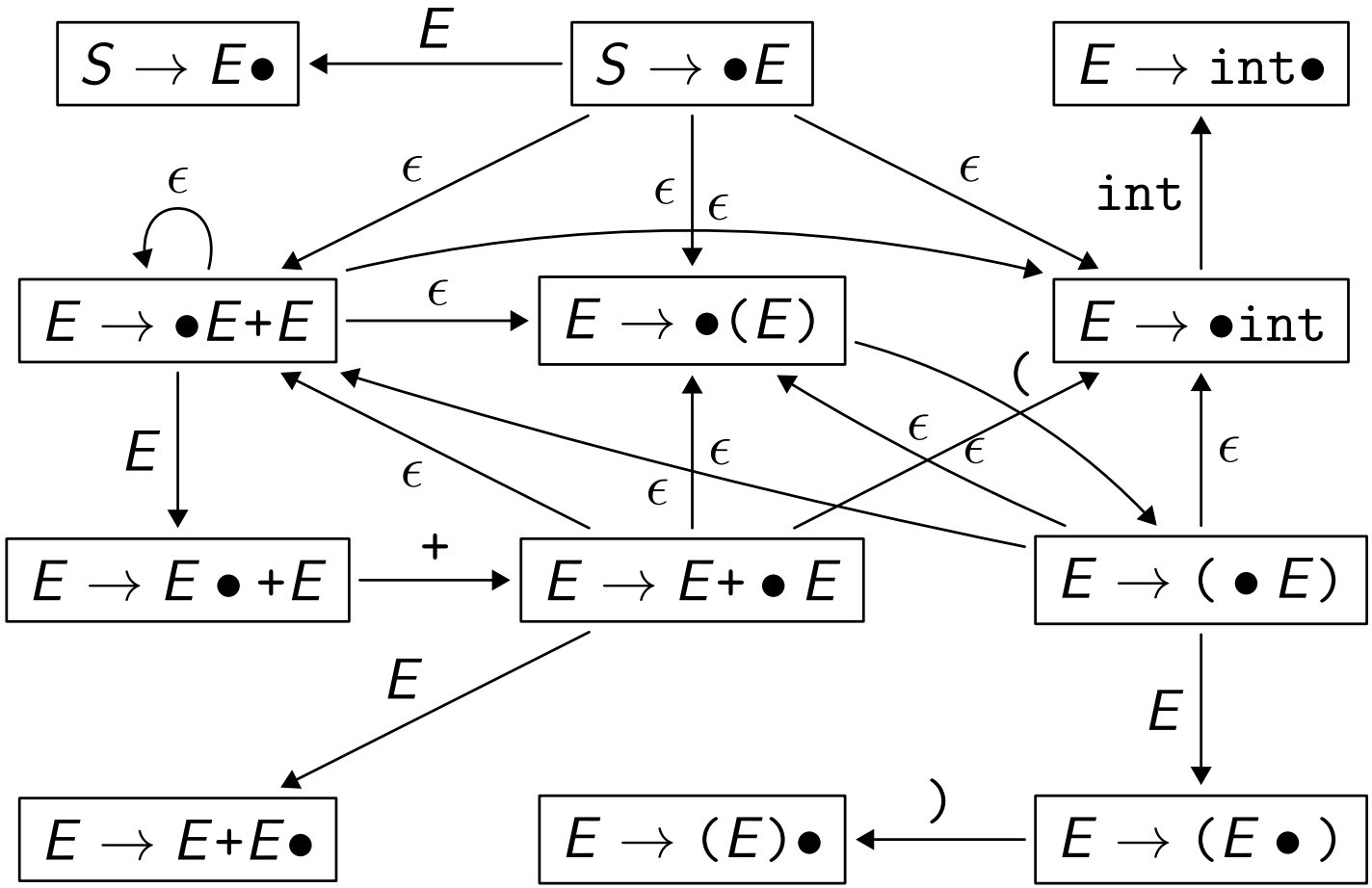
$$[X \to \alpha \bullet \beta]$$

où $X \to \alpha\beta$ 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 »

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

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

exemple



$$
\begin{array}{lcl}
S & \to & E \\
\\
E & \to & E{+}E \\
 & | & (E) \\
 & | & \texttt{int}
\end{array}
$$

déterminisons l'automate LR

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

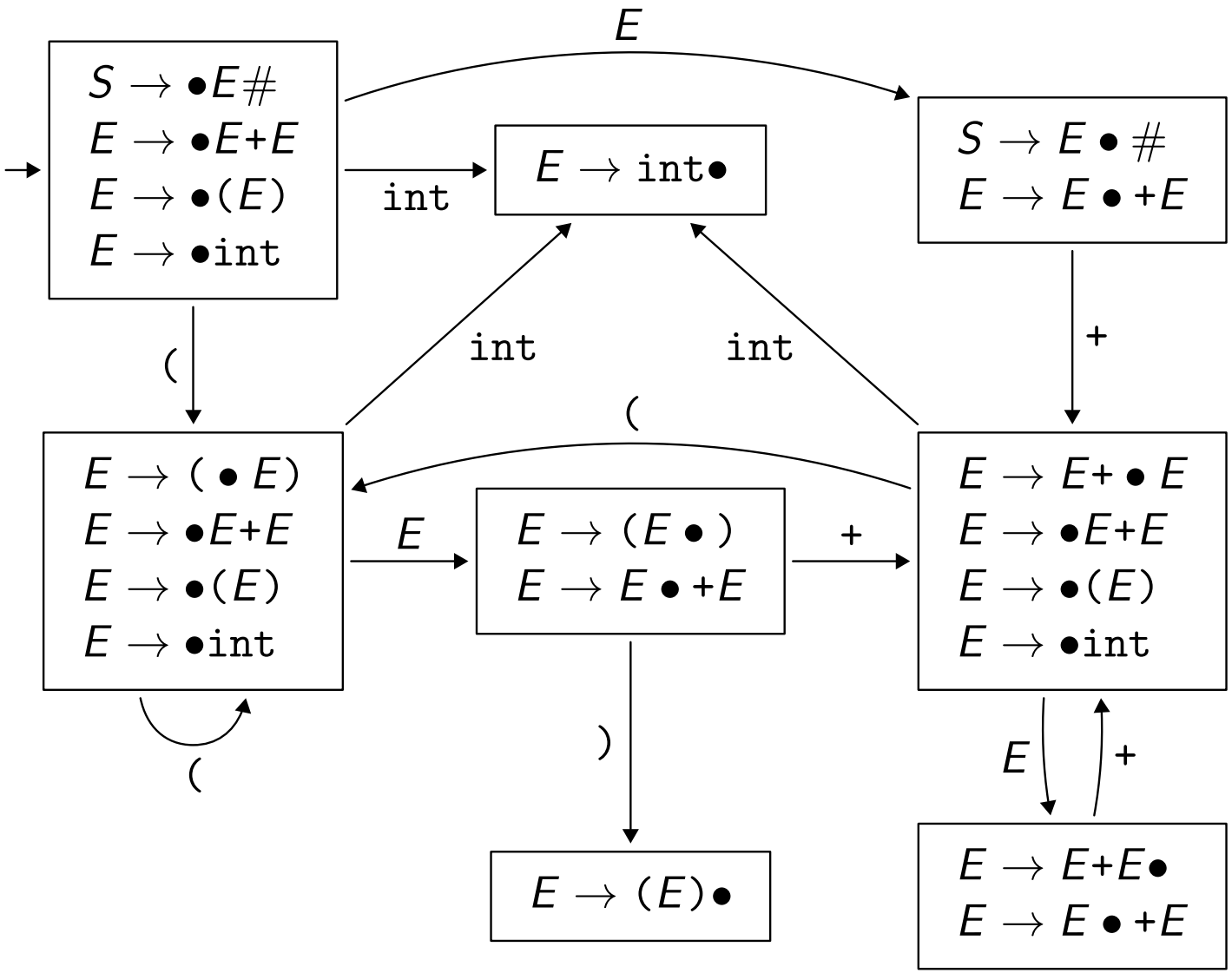
les états de l'automate déterministe sont donc des ensembles d'*items*, tels que

$$
\begin{array}{l}
E \rightarrow E\texttt{+} \bullet E \\
E \rightarrow \bullet E\texttt{+}E \\
E \rightarrow \bullet\texttt{(}E\texttt{)} \\
E \rightarrow \bullet\texttt{int}
\end{array}
$$

par construction, chaque état s est **saturé** par la propriété

$$\begin{array}{rl} \text{si} & Y \to \alpha \bullet X \beta \in s \\ \text{et si} & X \to \gamma \text{ est une production} \\ \text{alors} & X \to \bullet \gamma \in s \end{array}$$

l'état initial est celui contenant $S \to \bullet E \ \#$



$$
\begin{array}{lcl}
S & \to & E\ \# \\
E & \to & E{+}E \\
 & | & (E) \\
 & | & \texttt{int}
\end{array}
$$

on construit ainsi la table `action` :

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

on construit ainsi la table `goto` :

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

sur notre exemple, la table est la suivante :

	action					*goto*
état	(	)	+	`int`	#	E
1	`shift 4`			`shift 2`		3
2	`reduce` $E \rightarrow$ `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					

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.

ici, on a un conflit lecture/réduction dans l'état 8

$$\boxed{\begin{array}{l} E \to E{+}E\bullet \\ E \to E\bullet{+}E \end{array}}$$

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

en privilégiant la réduction (associativité à gauche), c'est-à-dire

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

dérouler l'analyse ascendante du mot `int+int+int` (11 étapes)

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

une idée très simple consiste à poser $\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ si et seulement si

$$[X \to \beta\bullet] \in s \quad \text{et} \quad a \in \textsc{follow}(X)$$

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

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

est SLR(1)

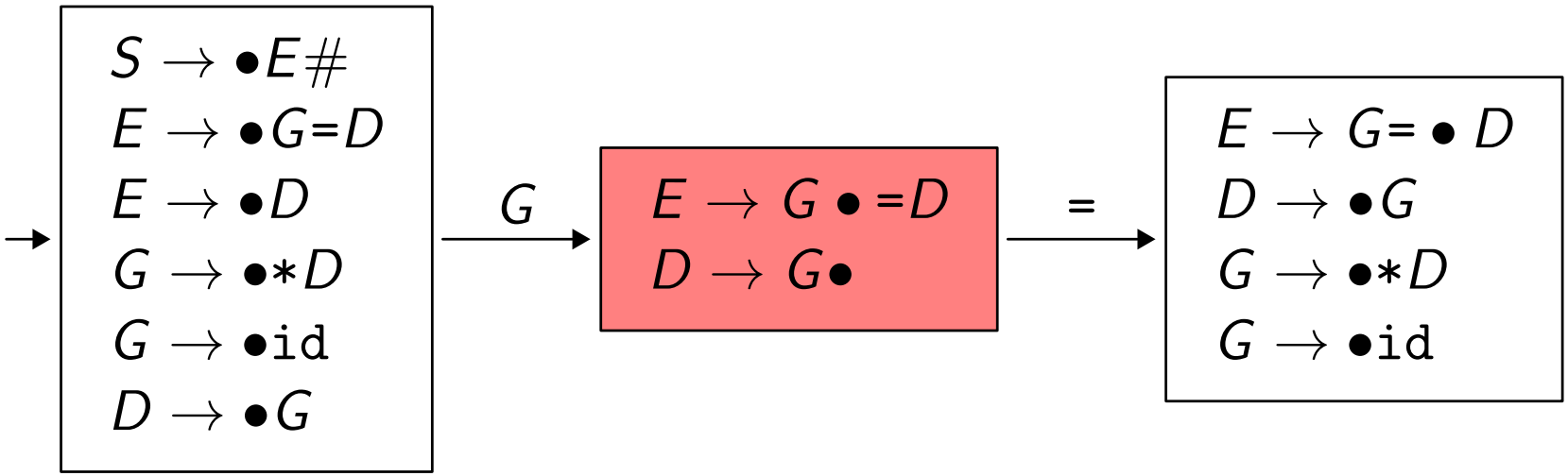
exercice : le vérifier (l'automate contient 12 états)

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

exemple :

$$
\begin{array}{lcl}
S & \to & E\# \\
E & \to & G = D \\
 & | & D \\
G & \to & * D \\
 & | & \texttt{id} \\
D & \to & G
\end{array}
$$

	=	
1	...	...
2	shift 3 reduce $D \to G$	...
3	⋮	⋱



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

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

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

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

$$
\begin{array}{rcl}
[Y \to \alpha \bullet a\beta, b] & \xrightarrow{a} & [Y \to \alpha a \bullet \beta, b] \\
[Y \to \alpha \bullet X\beta, b] & \xrightarrow{X} & [Y \to \alpha X \bullet \beta, b] \\
[Y \to \alpha \bullet X\beta, b] & \xrightarrow{\epsilon} & [X \to \bullet\gamma, c] \quad \text{pour tout } c \in \textsc{first}(\beta b)
\end{array}
$$

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

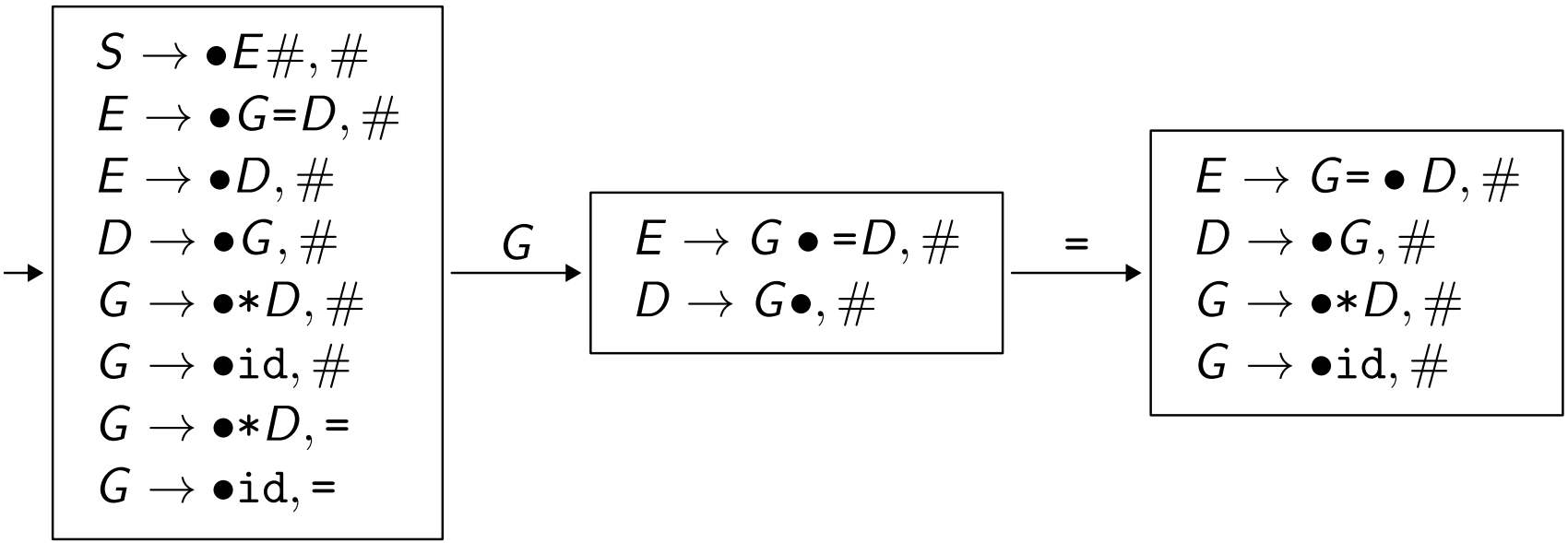
comme précédemment, on peut déterminiser l'automate et construire la table correspondante ; on introduit une action de réduction pour (s, a) seulement lorsque s contient un item de la forme $[X \to \alpha\bullet, a]$

Définition (classe LR(1))

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

$$
\begin{array}{lcl}
S & \to & E\# \\
E & \to & G = D \\
 & | & D \\
G & \to & * D \\
 & | & \texttt{id} \\
D & \to & G
\end{array}
$$

	#	=	
1	...	...	...
2	reduce $D \to G$	shift 3	...
3	⋮	⋮	⋱

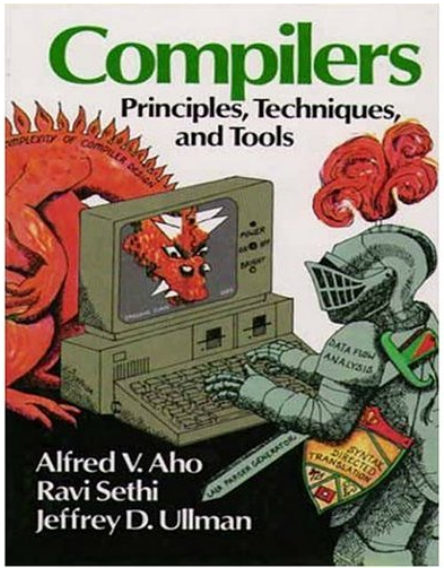


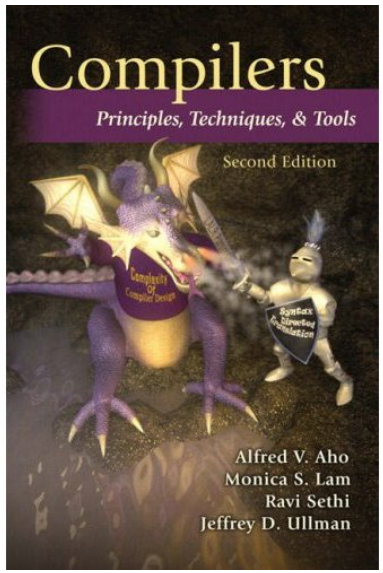
LALR(1)

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

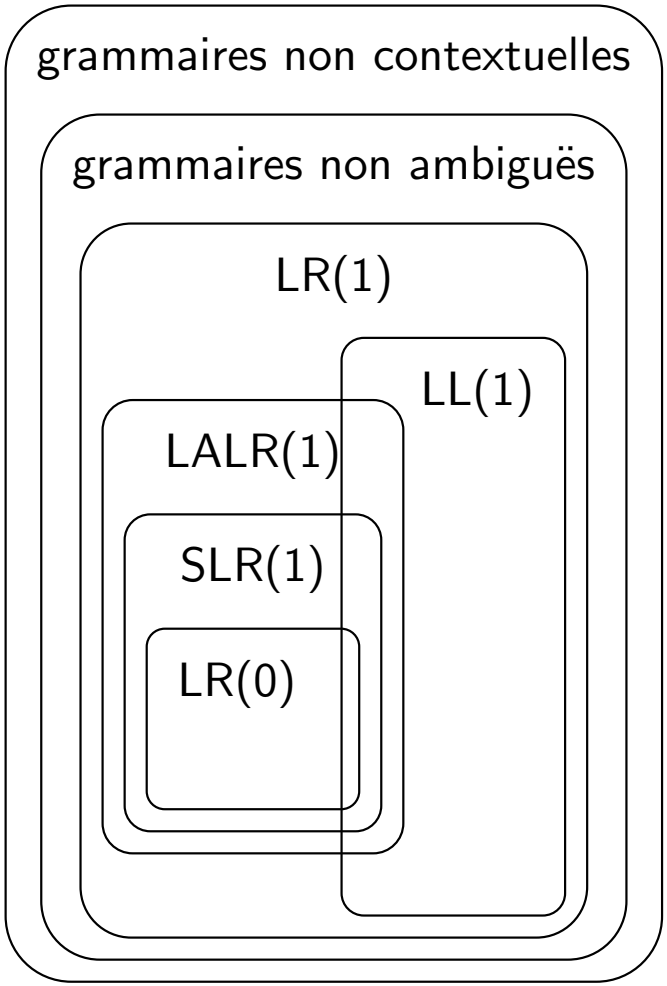
la classe LALR(1) (*lookahead LR*) est une telle approximation, utilisée notamment dans beaucoup d'outils de la famille `yacc`

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

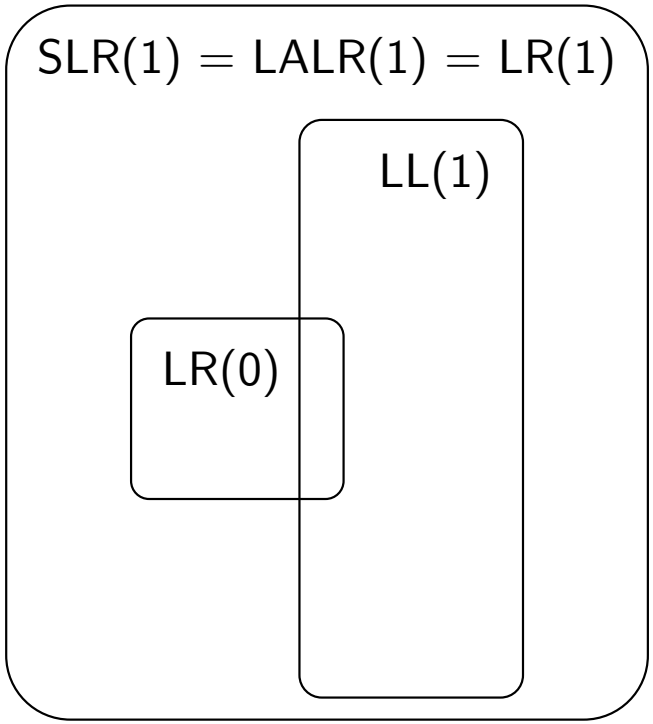




grammaires

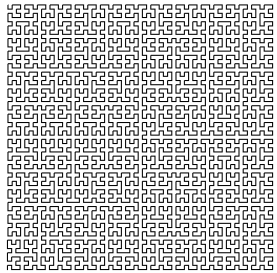
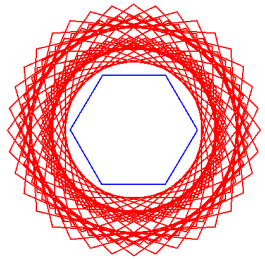
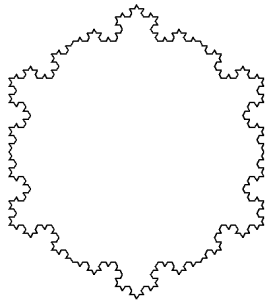
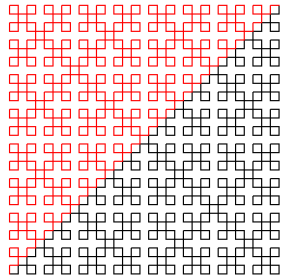


langages



- TD 4

  utilisation d'ocamllex + menhir
  sur un petit langage Logo
  (tortue graphique)

- prochain cours
  - analyse syntaxique 2/2