École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 7 / 18 novembre 2016

résumé de l'épisode précédent

- notion de grammaire
  - dérivation
  - grammaire ambiguë
- analyse descendante
  - une table nous indique quelle expansion choisir
  - calcul de NULL, FIRST, FOLLOW
    - calcul de point fixe
  - grammaire LL(1)

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

analyse ascendante

idée générale

l'idée est toujours de lire l'entrée de gauche à droite, mais on cherche maintenant à reconnaître des membres droits de productions pour construire l'arbre de dérivation de bas en haut (*bottom-up parsing*)

l'analyse manipule toujours 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

exemple

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

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 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

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

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

soit a le premier caractère 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 $s_n \xrightarrow{a} s$
- 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 $s_{n-p} \xrightarrow{X} s$, *i.e.*

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

construction de l'automate et de la table

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

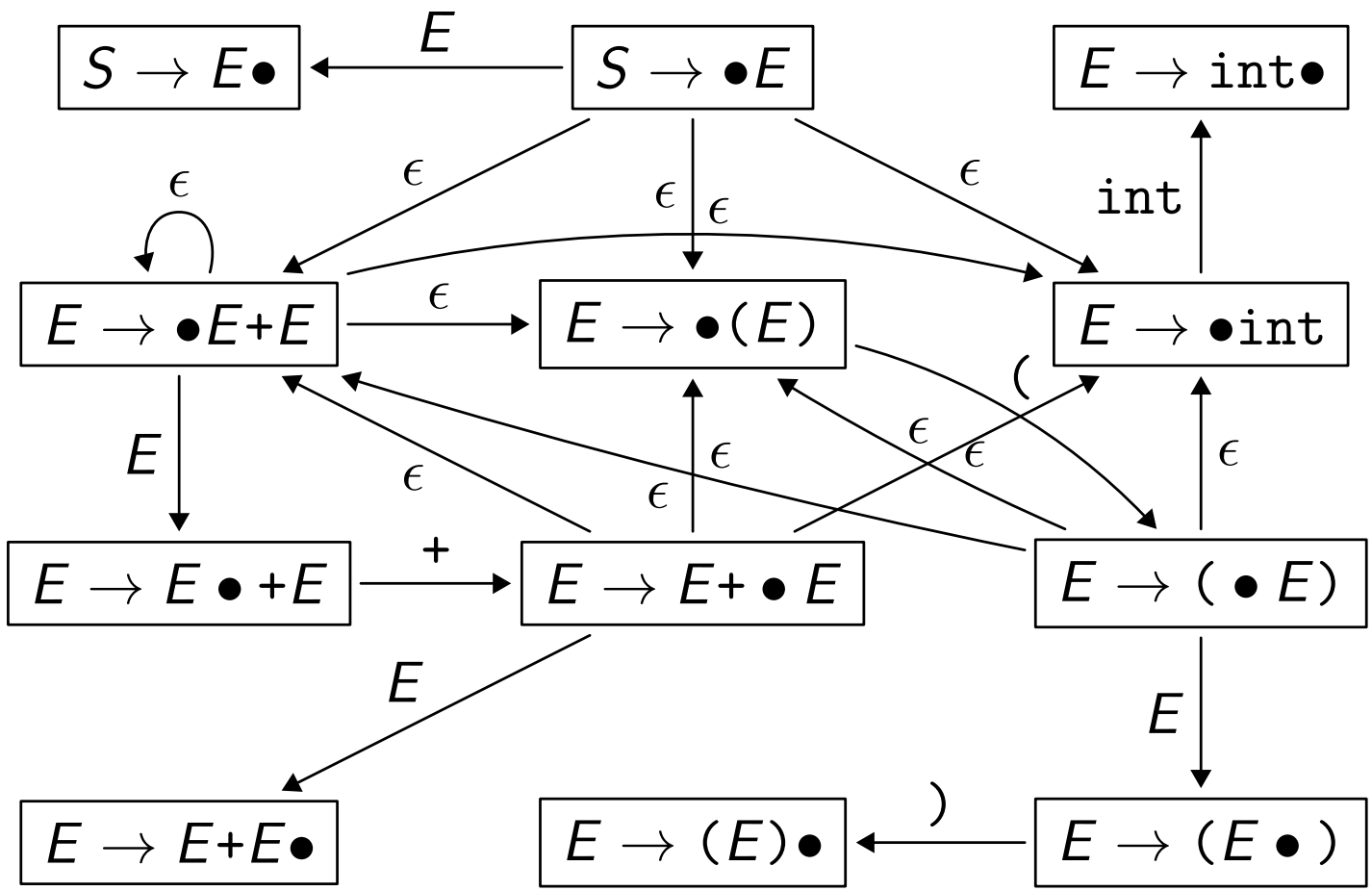
$$[X \to \alpha \bullet \beta]$$

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

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

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



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

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

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

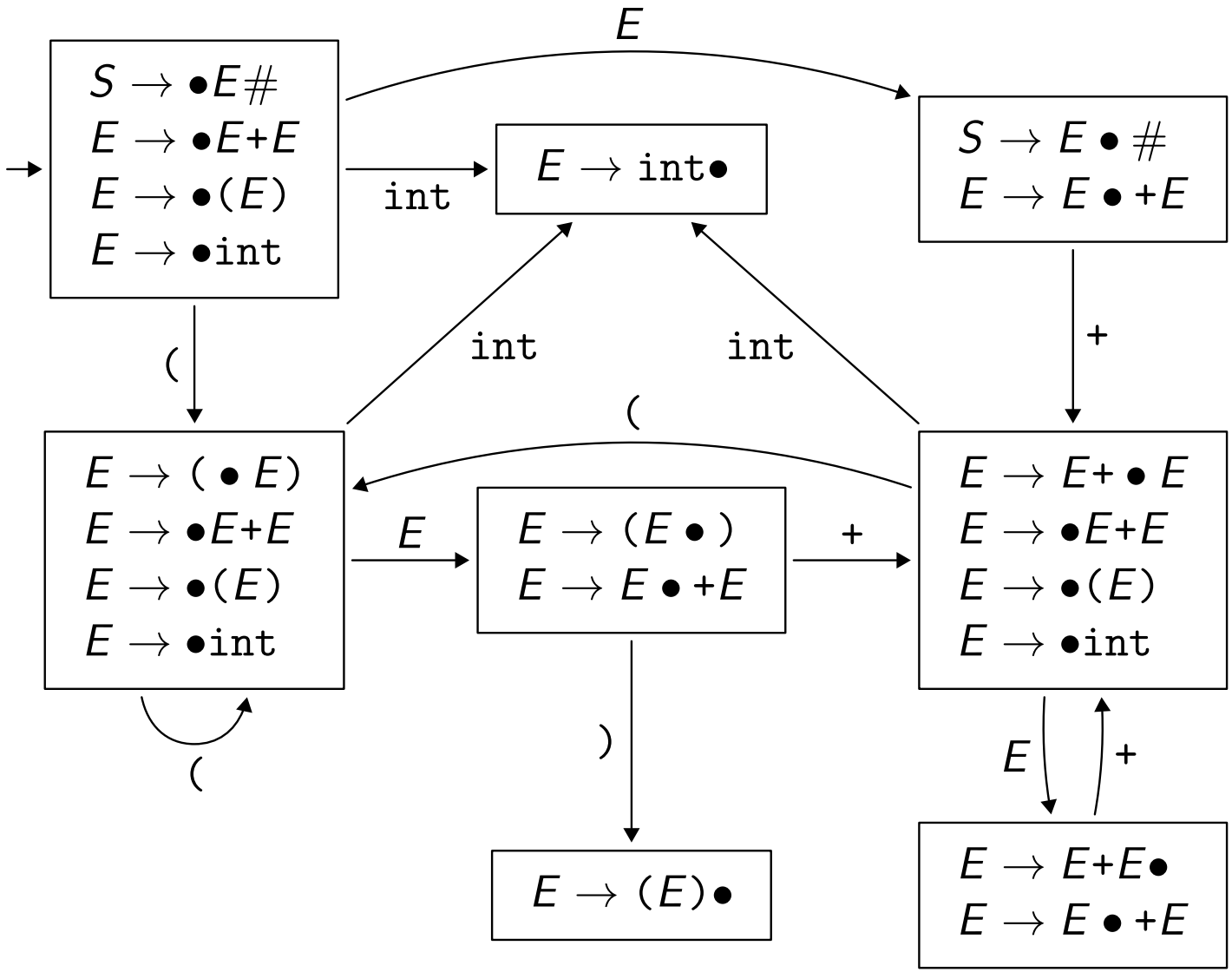
$$\begin{array}{rl} \text{si} & Y \to \alpha \bullet X \beta \in s \\ \text{et si} & X \to \gamma \text{ est une production} \\ \text{alors} & X \to \bullet \gamma \in s \end{array}$$

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

exemple

$$S \rightarrow E$$

$$E \rightarrow E{+}E \mid (E) \mid \texttt{int}$$



en pratique, on travaille 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

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.

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 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 $E \to$ 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					

pile	entrée	action
1	int+int+int	s2
1 int 2	+int+int	$E \to$ int, g3
1 E 3	+int+int	s6
1 E 3 + 6	int+int	s2
1 E 3 + 6 int 2	+int	$E \to$ int, g8
1 E 3 + 6 E 8	+int	$E \to E$+E, g3
1 E 3	+int	s6
1 E 3 + 6	int	s2
1 E 3 + 6 int 2	#	$E \to$ int, g8
1 E 3 + 6 E 8	#	$E \to E$+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 $\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ 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)

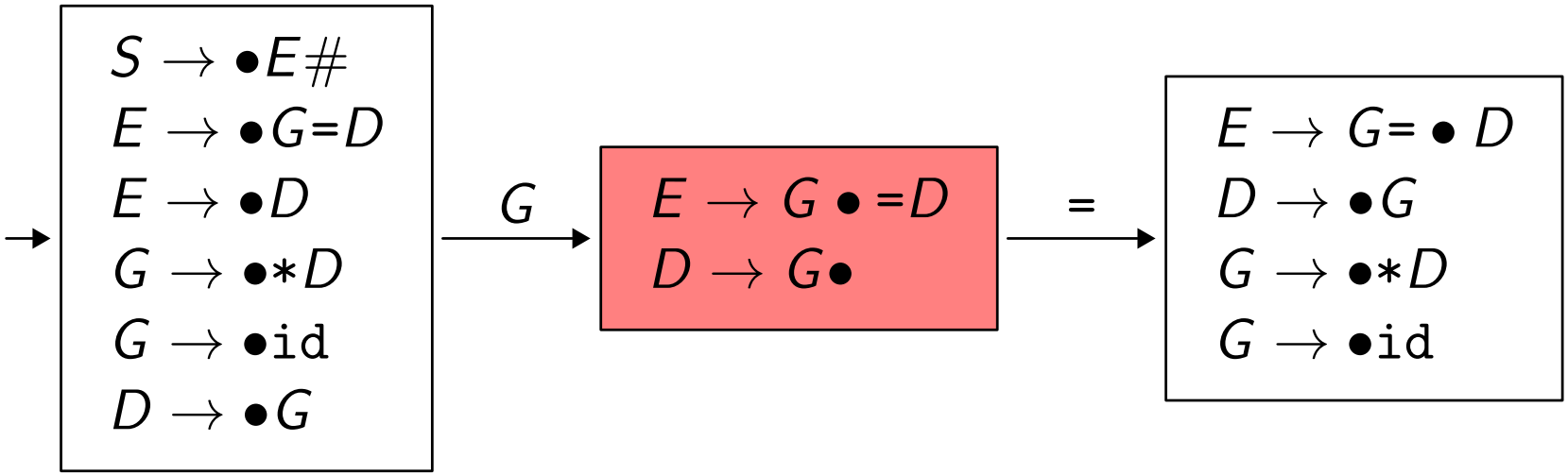
exercice : le vérifier (l'automate contient 12 états)

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 caractère 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 $[S \to \bullet\alpha, \#]$

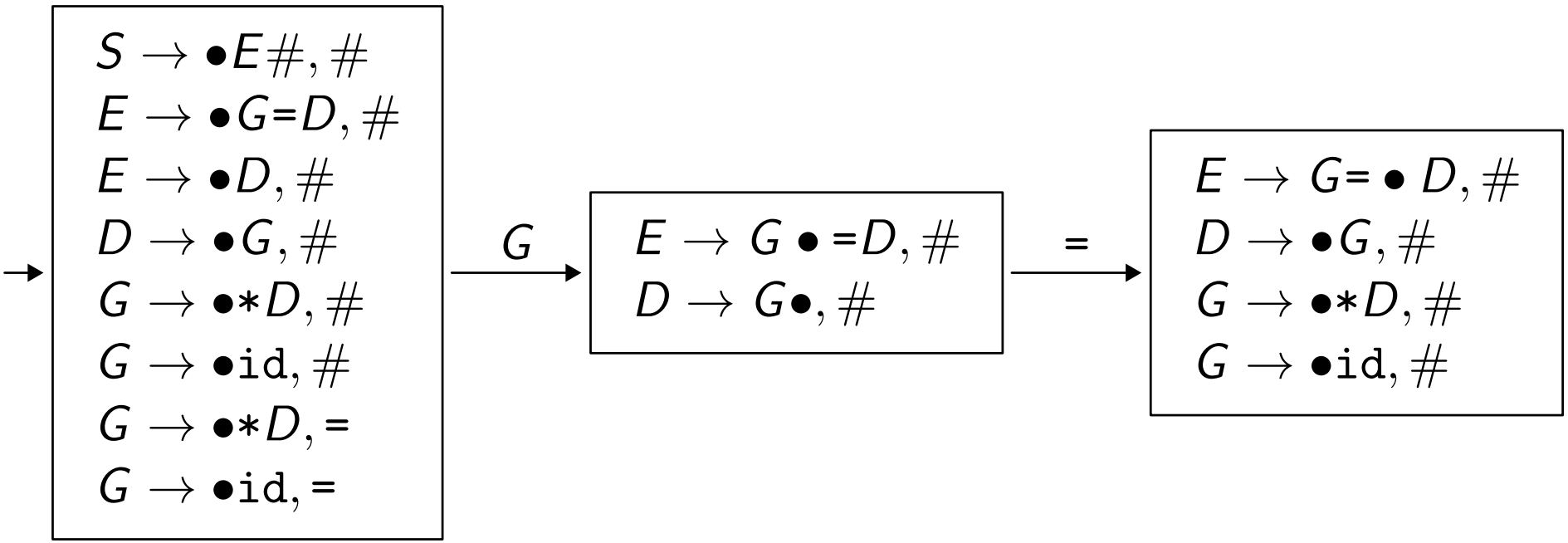
comme précédemment, on peut déterminiser l'automate et construire la table correspondante ; on introduit une action de réduction pour (s, a) seulement lorsque s contient un item de la forme $[X \to \alpha\bullet, a]$

Définition (classe LR(1))

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

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

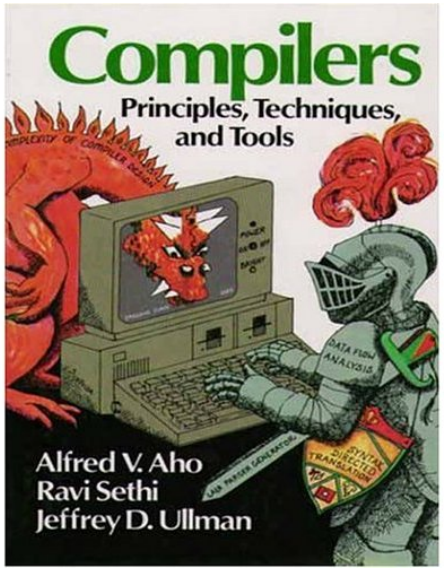
	#	=	
1	...	...	...
2	reduce $D \to G$	shift 3	...
3	⋮	⋮	⋱

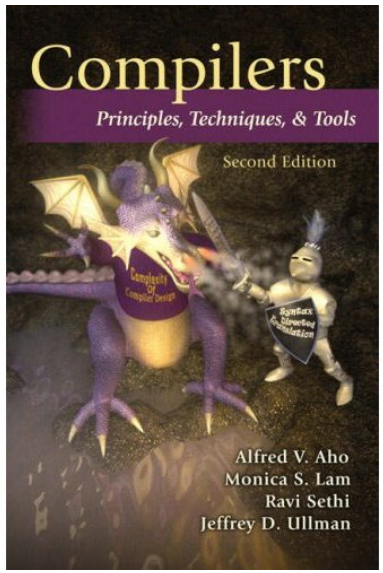


la construction LR(1) pouvant être coûteuse, il existe des approximations

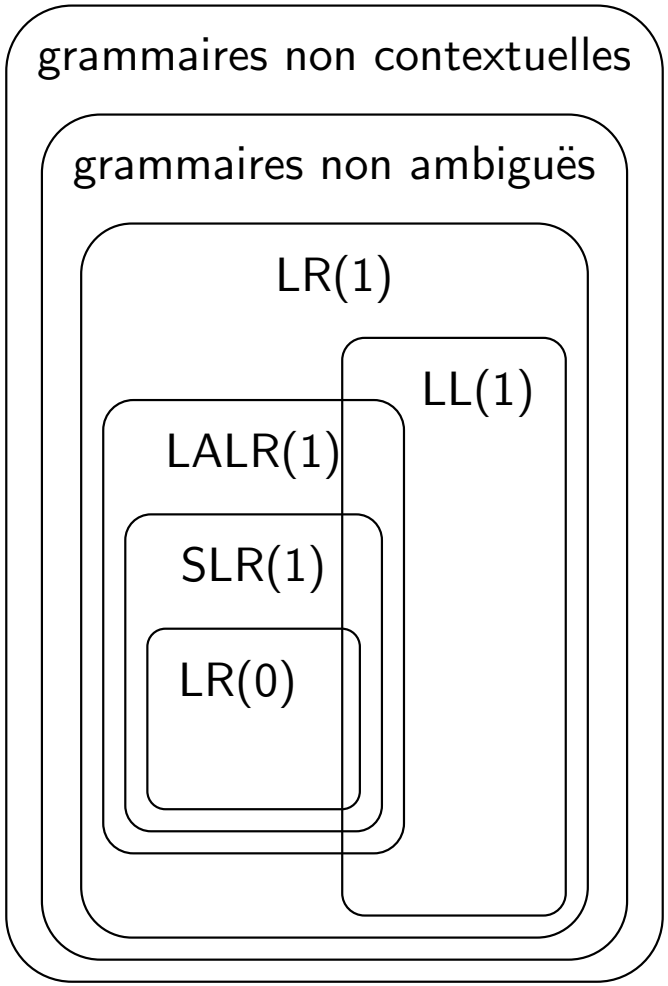
la classe LALR(1) (*lookahead LR*) est une telle approximation, utilisée notamment dans les outils de la famille `yacc`

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

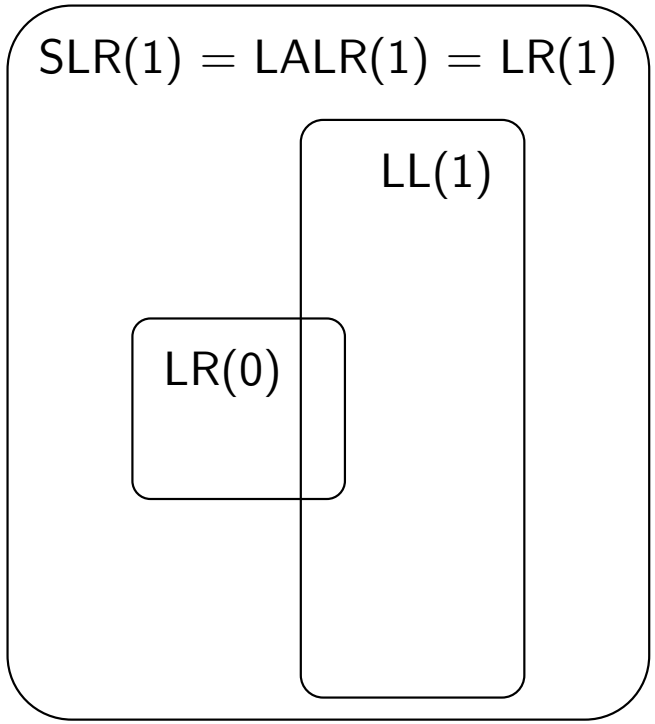




grammaires



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`, `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 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
```

- 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 5)

quand on combine `ocamllex` et `menhir`

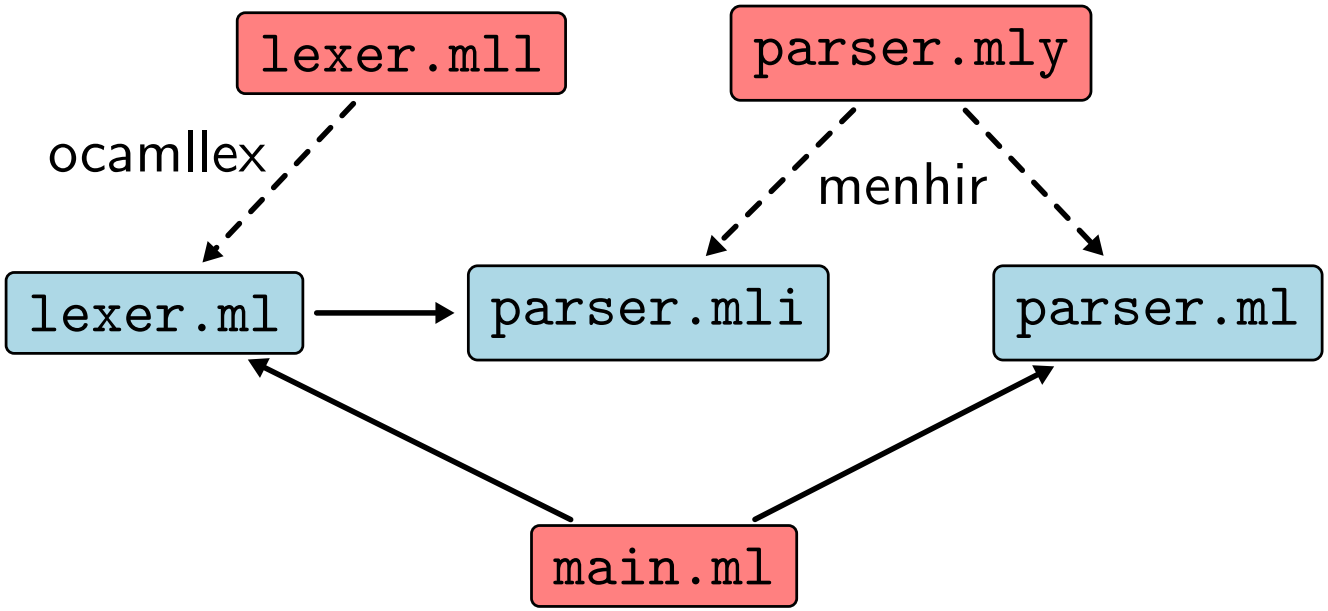
- `lexer.mll` fait référence aux lexèmes définis dans `parser.mly`

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

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

```
let c  = open_in file in
let lb = Lexing.from_file 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
```

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
    { ... }
| ...
```

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
```

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

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 ($E?$, $E*$, $E+$), des listes avec séparateur
- explication des conflits
- mode interactif
- analyse LR(1) plutôt 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 un traitement spécial du retour chariot dans l'analyseur lexical (voir par exemple `lexer.mll` fourni dans le TD 3)

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
  | ...
```

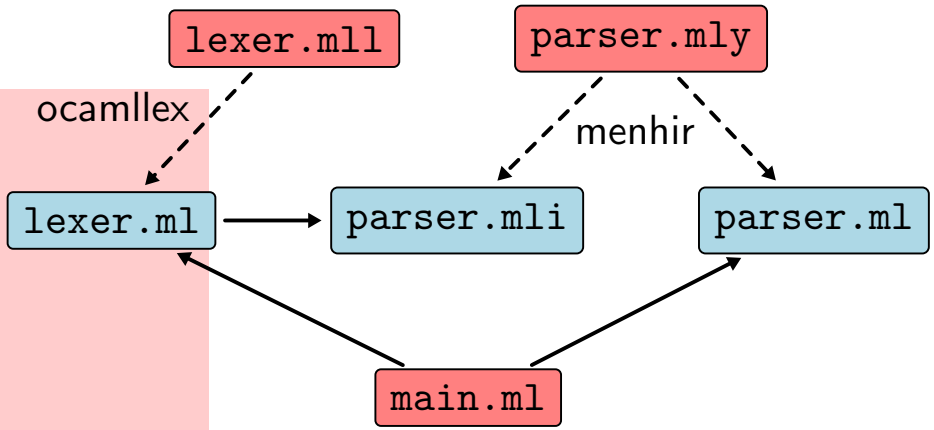
la grammaire peut donc ressembler à ceci

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

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

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

le `Makefile` ressemble donc à ceci :

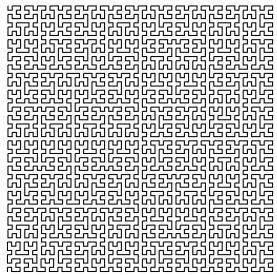
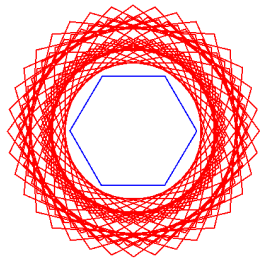
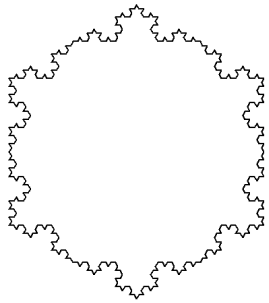
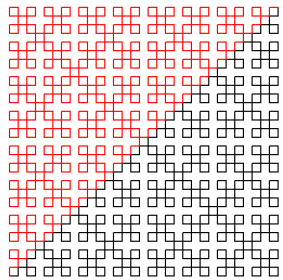
```
lexer.ml: lexer.mll
        ocamllex lexer.mll

parser.mli parser.ml: parser.mly
        menhir -v parser.mly

.depend: lexer.ml parser.mli parser.ml
        ocamldep *.ml *.mli > .depend

include .depend
```



(cf. `Makefile` fourni avec le TD 2 ou 3 / sinon utiliser `ocamlbuild`)

- TD 7

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

- prochain cours
  - production de code