École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 6 / 10 novembre 2011

Petite devinette

quel point commun ?

Analyse syntaxique

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

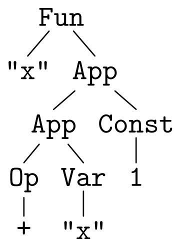
Analyse syntaxique

suite de lexèmes

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

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

Quels outils

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

on suppose connues les notions de grammaire, dérivation et automate à pile ; cf. cours langages formels
dans ce qui suit, on note T l'ensemble des terminaux, N l'ensemble des non terminaux et S le symbole de départ d'une grammaire donnée

Exemple

pour un langage d'expressions arithmétiques construites à partir de constantes entières, d'additions, de multiplications et de parenthèses, on se donne la grammaire suivante :

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

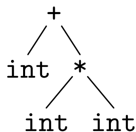
les terminaux de la grammaire sont les lexèmes produits par l'analyse lexicale, ici $T = \{\texttt{+}, \ast, \texttt{(}, \texttt{)}, \texttt{int}\}$

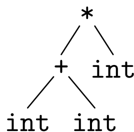
note : `int` désigne le lexème correspondant à une constante entière (*i.e.* sa nature, pas sa valeur)

Ambiguïté

cette grammaire est **ambiguë** car certains mots admettent plusieurs arbres de dérivation

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





Grammaire non ambiguë

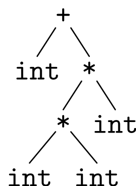
pour ce langage-là, il est néanmoins possible de proposer une autre grammaire, non ambiguë, qui définit le même langage

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

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

Grammaire non ambiguë

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



correspondant à la dérivation gauche

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

Objectif

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

on va donc se restreindre à des classes de grammaires non ambiguës, et où on sait en outre construire un automate à pile **déterministe** correspondant

ces classes s'appellent LL(1), LR(0), SLR(1), LALR(1), LR(1), etc.

analyse descendante

Idée générale

idée : procéder par expansions successives du non terminal le plus à gauche (on construit donc une dérivation gauche) en partant de S et en se servant d'une **table** indiquant, pour un non terminal X à expanser et les k premiers caractères de l'entrée, l'expansion $X \rightarrow \beta$ à effectuer
(en anglais on parle de *top-down parsing*)

supposons $k = 1$ par la suite et notons $T(X, c)$ cette table

en pratique on suppose qu'un symbole terminal $\#$ dénote la fin de l'entrée, et la table indique donc également l'expansion de X lorsque l'on atteint la fin de l'entrée

Fonctionnement de l'analyse

on utilise une pile qui est un mot de $(N \cup T)^\star$; initialement la pile est réduite au symbole de départ

à chaque instant, on examine le sommet de la pile et le premier caractère c de l'entrée

- si la pile est vide, on s'arrête ; il y a succès si et seulement si c est $\#$
- si le sommet de la pile est un terminal a, alors a doit être égal à c, on dépile a et on consomme c ; sinon on échoue
- si le sommet de la pile est un non terminal X, alors on remplace X par le mot $\beta = T(X, c)$ en sommet de pile, le cas échéant, en empilant les caractères de β en partant du dernier ; sinon, on échoue

Exemple

transformons encore la grammaire des expressions arithmétiques
et considérons la table d'expansion suivante

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

	+	*	(	)	int	#
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		int	

Exemple

illustrons l'analyse descendante du mot

`int + int * int`

	+	*	(	)	int	#
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		int	

pile	entrée
E	int+int*int#
$E'T$	int+int*int#
$E'T'F$	int+int*int#
$E'T'$int	int+int*int#
$E'T'$	+int*int#
E'	+int*int#
$E'T+$	+int*int#
$E'T$	int*int#
$E'T'F$	int*int#
$E'T'$int	int*int#
$E'T'$	*int#
$E'T'F*$	*int#
$E'T'F$	int#
$E'T'$int	int#
$E'T'$	#
E'	#
ϵ	#

Programmation d'un analyseur descendant

un analyseur descendant se programme très facilement en introduisant une fonction pour chaque non terminal de la grammaire

chaque fonction examine l'entrée et, selon le cas, la consomme ou appelle récursivement les fonctions correspondant à d'autres non terminaux, selon la table d'expansion

Programmation d'un analyseur descendant

faisons le choix d'une programmation purement applicative, où l'entrée est une liste de lexèmes du type

```
type token = Tplus | Tmult | Tleft | Tright | Tint | Teof
```

on va donc construire cinq fonctions qui « consomment » la liste d'entrée

```
val e : token list → token list
val e' : token list → token list
val t : token list → token list
val t' : token list → token list
val f : token list → token list
```

et la reconnaissance d'une entrée pourra alors se faire ainsi

```
let recognize l = e l = [Teof];;
```

Programmation d'un analyseur descendant

les fonctions procèdent par filtrage sur l'entrée, et suivent la table

	+	*	(	)	int	#
E			TE'		TE'	

```
let rec e = function
  | (Tleft | Tint) :: _ as m → e' (t m)
  | _ → error ()
```

	+	*	(	)	int	#
E'	$+TE'$			ϵ		ϵ

```
and e' = function
  | Tplus :: m → e' (t m)
  | (Tright | Teof) :: _ as m → m
  | _ → error ()
```

Programmation d'un analyseur descendant

	+	*	(	)	int	#
T			FT'		FT'	

```
and t = function
  | (Tleft | Tint) :: _ as m → t' (f m)
  | _ → error ()
```

	+	*	(	)	int	#
T'	ϵ	$*FT'$		ϵ		ϵ

```
and t' = function
  | (Tplus | Tright | Teof) :: _ as m → m
  | Tmult :: m → t' (f m)
  | _ → error ()
```

Programmation d'un analyseur descendant

	+	*	(	)	int	#
F			(E)		int	

```
and f = function
  | Tint :: m -> m
  | Tleft :: m -> begin match e m with
      | Tright :: m -> m
      | _ -> error ()
    end
  | _ -> error ()
```

Programmation d'un analyseur descendant

remarques

- la table d'expansion n'est pas explicite : elle est dans le code de chaque fonction
- la pile non plus n'est pas explicite : elle est réalisée par la pile d'appels
- on aurait pu les rendre explicites

- on aurait pu aussi faire le choix d'une programmation plus impérative

```
val next_token : unit → token
```

Construire la table d'expansion

reste une question d'importance : comment construire la table ?

l'idée est simple : pour décider si on réalise l'expansion $X \to \beta$ lorsque le premier caractère de l'entrée est c, on va chercher à déterminer si c fait partie des **premiers** caractères des mots reconnus par β

une difficulté se pose pour une production telle que $X \to \epsilon$, et il faut alors considérer aussi l'ensemble des caractères qui peuvent **suivre** X

déterminer les premiers et les suivants nécessite également de déterminer si un mot peut se dériver en ϵ

Définitions

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.* $\alpha \xrightarrow{\star} \epsilon$.

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.* $\{a \in T \mid \exists w.\, \alpha \xrightarrow{\star} aw\}$.

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.* $\{a \in T \mid \exists u, w.\, S \xrightarrow{\star} uXaw\}$.

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 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

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

Calcul de point fixe

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 $f : A \to A$ monotone, 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 monotone, 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

Calcul de NULL

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

pour calculer les $\text{NULL}(X_i)$, on part donc de $\text{NULL}(X_1) = \texttt{false}$, ..., $\text{NULL}(X_n) = \texttt{false}$ et on applique les équations jusqu'à obtention du point fixe

Exemple

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

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

Calcul de FIRST

de même, les équations définissant FIRST sont mutuellement récursives

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

et

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

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

Exemple

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

NULL

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

FIRST

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

Calcul de FOLLOW

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 introduite $\#$ dans les suivants du symbole de départ
(ce que l'on peut faire directement, ou en ajoutant une règle $S' \to S\#$)

Exemple

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

NULL

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

FIRST

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

FOLLOW

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

Application : construction de la table d'expansion

à l'aide des premiers et des suivants, on construit la table d'expansion $T(X, a)$ de la manière suivante

pour chaque production $X \to \beta$,

- on pose $T(X, a) = \beta$ pour tout $a \in \text{FIRST}(\beta)$
- si $\text{NULL}(\beta)$, on pose aussi $T(X, a) = \beta$ pour tout $a \in \text{FOLLOW}(X)$

Exemple

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

FIRST

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

FOLLOW

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

	+	*	(	)	int	#
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		int	

Grammaire LL(1)

Définition (grammaire LL(1))

Une grammaire est dite LL(1) si, dans la table précédente, il y a au plus une production dans chaque case.

LL signifie « **L**eft to right scanning, **L**eftmost derivation »

Construire des grammaires LL(1)

il faut souvent transformer les grammaires pour les rendre LL(1)

en particulier, une grammaire **récursive gauche**, *i.e.* contenant une production de la forme

$$X \to X\alpha$$

ne sera jamais LL(1)

il faut alors supprimer la récursion gauche (directe ou indirecte)

de même il faut factoriser les productions qui commencent par le même terminal (factorisation gauche)

Conclusion

les analyseurs LL(1) sont relativement simples à écrire

mais ils nécessitent d'écrire des grammaires peu naturelles

on va se tourner vers une autre solution...

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*)

Fonctionnement de l'analyse

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

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

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

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

Analyse LR

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 $X \to \alpha$, avec α de longueur p, alors on doit trouver α en sommet de pile

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

  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.*

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

Analyse LR

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` $X \to \alpha$ pour une réduction
  - un succès
  - un échec

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

Automate LR(0)

commençons par $k = 0$

on construit un automate LR(0) non déterministe

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

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

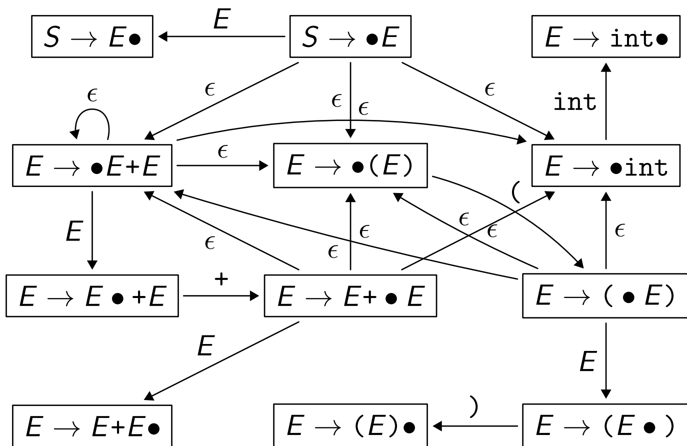
où $X \rightarrow \alpha\beta$ est une production de la grammaire ; l'intuition est : « je cherche à reconnaître X, j'ai déjà lu α et je dois encore lire β »

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

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

Exemple

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



Automate LR(0) déterministe

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

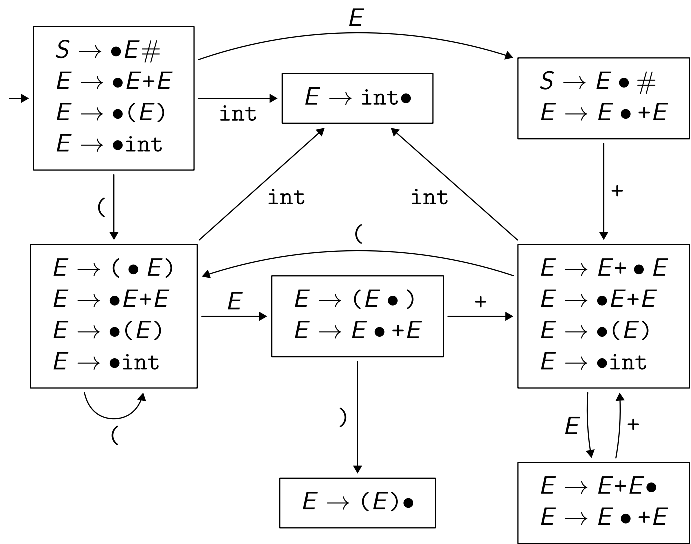
$$\begin{array}{|l|} E \rightarrow E\texttt{+}\bullet E \\ E \rightarrow \bullet E\texttt{+}E \\ E \rightarrow \bullet\texttt{(}E\texttt{)} \\ E \rightarrow \bullet\texttt{int} \end{array}$$

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

Exemple

$$S \rightarrow E$$

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



Construction de la table

on construit ainsi la table `action`

- $\texttt{action}(s, \#) = $ succès si $[S \rightarrow E \bullet \#] \in s$
- $\texttt{action}(s, a) = \texttt{shift}\ s'$ si on a une transition $s \xrightarrow{a} s'$
- $\texttt{action}(s, a) = \texttt{reduce}\ X \rightarrow \beta$ si $[X \rightarrow \beta \bullet] \in s$, 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 $s \xrightarrow{X} s'$

Conflits

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.

Exemple

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					

Conflit

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

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

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

Exemple d'exécution

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

Analyse SLR(1)

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

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

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

Définition (classe SLR(1))

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

(SLR signifie *Simple LR*)

Exemple

la grammaire

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

est SLR(1)

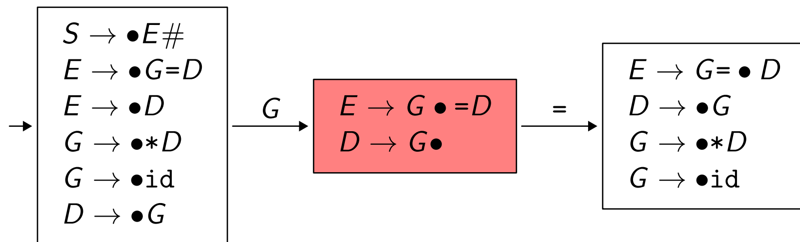
exercice : le vérifier (l'automate contient 12 états)

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

exemple :

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

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



Analyse LR(1)

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 »

Analyse LR(1)

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

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

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

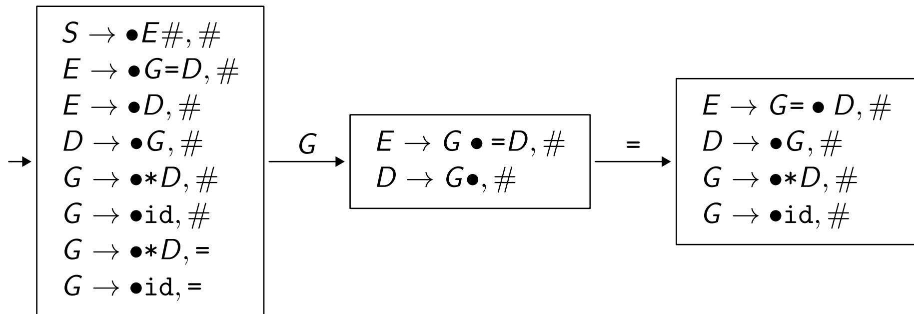
Définition (classe LR(1))

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

Exemple

$$
\begin{array}{lcl}
S & \rightarrow & E\# \\
E & \rightarrow & G = D \\
  & \mid & D \\
G & \rightarrow & * D \\
  & \mid & \mathtt{id} \\
D & \rightarrow & G
\end{array}
$$

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

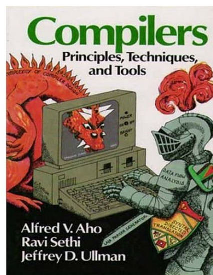


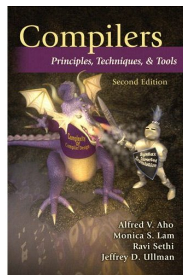
LALR(1)

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

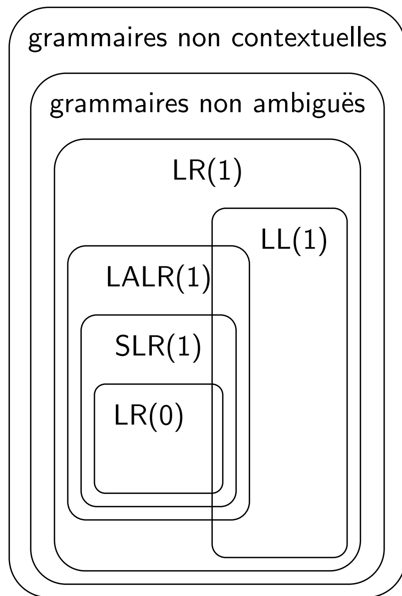
la classe LALR(1) (*lookahead LR*) est une telle approximation, utilisée notamment dans les outils de la famille `yacc`

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





Hiérarchie



Conclusion

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`, `cups`, `menhir`, ...
(YACC signifie *Yet Another Compiler Compiler*)

l'outil Menhir

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`)

Structure

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 ...
```

Exemple

```
%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 }
;
```

Exemple

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

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

on obtient du code OCaml pur dans `arith.ml`, 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)

Conflits

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

Exemple

sur la grammaire ci-dessus, Menhir signale un conflit

```
% menhir -v arith.mly && ocamlc -c -i arith.ml
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
```

Exemple

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
```

Résolution des conflits

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

Résolution des conflits

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

Exemple

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 }
;
```

Priorités

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
```

Facilités

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 !

Localisations

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)

Localisation

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

```
type expression =
  { expr_desc: expr_desc;
    expr_loc : Lexing.position × Lexing.position }

and expr_desc =
  | Econst of int
  | Eplus  of expression × expression
  | Eneg   of expression
  | ...
```

Compilation et dépendances

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`)

Plus d'information

pour plus d'informations, consulter le manuel de Menhir, accessible depuis la page du cours

La suite

- TD de mercredi
  - analyseur LL(1)

- Cours de jeudi
  - production de code