École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

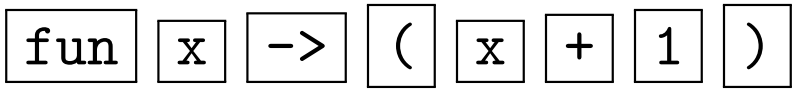
Cours 6 / 10 novembre 2017

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

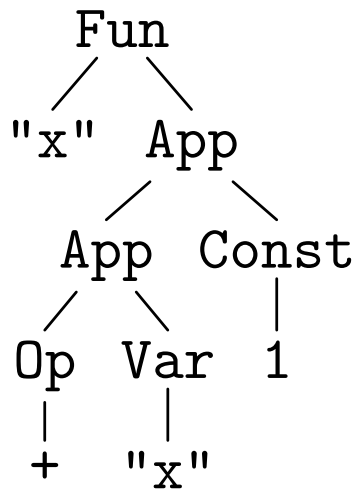


↓

analyse syntaxique

↓

syntaxe abstraite



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

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

pour l'analyse syntaxique, on va utiliser

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

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

Définition

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

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

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

en particulier

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

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

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

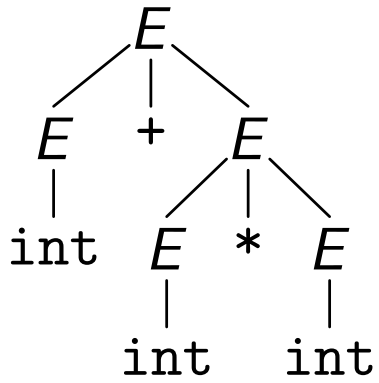
- *la racine est S*
- *les feuilles forment le mot w dans l'ordre infixe*
- *tout nœud interne X est un non terminal dont les fils sont étiquetés par $\beta \in (N \cup T)^\star$ avec $X \to \beta$ une règle de la dérivation*

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

la dérivation gauche

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

donne l'arbre de dérivation



mais la dérivation droite

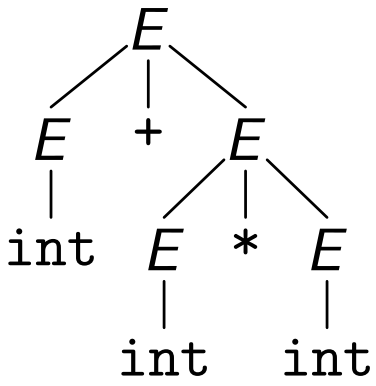
$$E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * \texttt{int} \rightarrow E + \texttt{int} * \texttt{int} \rightarrow \texttt{int} + \texttt{int} * \texttt{int}$$

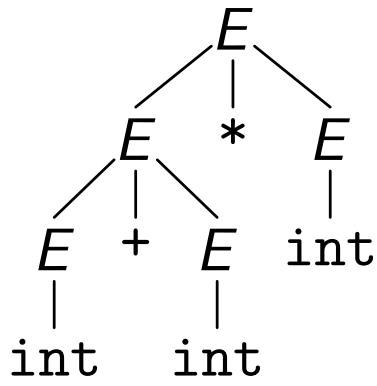
également

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



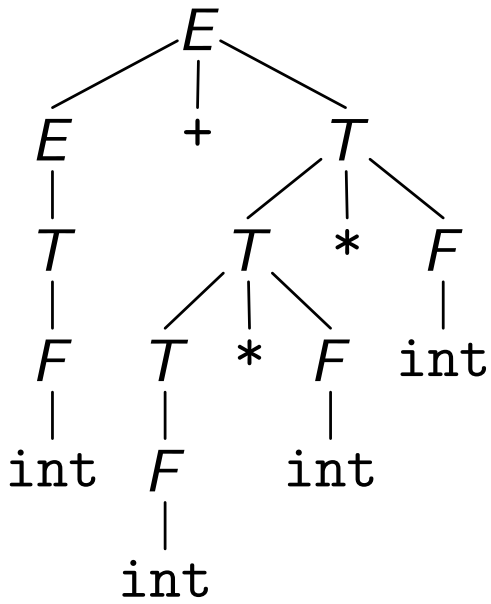


pour ce langage-là, il est néanmoins possible de proposer une autre grammaire, non ambiguë, qui définit le même langage

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

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

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

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

analyse descendante

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 $X \to \beta$ 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

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

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

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

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

(on verra plus loin comment construire cette table)

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	#

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

(c'est ce que nous avons fait au début du cours)

programmation d'un analyseur descendant

faisons le choix d'une programmation purement applicative, pour changer, 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]
```

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 ()
```

	+	*	(	)	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 ()
```

	+	*	(	)	int	#
F			(E)		int	

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

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

- en pratique, il faut aussi construire l'arbre de syntaxe abstraite

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

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

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 : le théorème de Tarski donne 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

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

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

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

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

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

NULL

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

FIRST

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

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 $S' \to S\#$)

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

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
{#}	∅	∅	∅	∅
{#, `)`}	{#}	{`+`, #}	∅	{`*`}
{#, `)`}	{#, `)`}	{`+`, #, `)`}	{`+`, #}	{`*`, `+`, #}
{#, `)`}	{#, `)`}	{`+`, #, `)`}	{`+`, #, `)`}	{`*`, `+`, #, `)`}
{#, `)`}	{#, `)`}	{`+`, #, `)`}	{`+`, #, `)`}	{`*`, `+`, #, `)`}

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 $a \in \text{FOLLOW}(X)$

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

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	

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 »

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)

les analyseurs LL(1) sont relativement simples à écrire
(cf TD de cette semaine)

mais ils nécessitent d'écrire des grammaires peu naturelles

on va se tourner vers une autre solution (la semaine prochaine)

la suite

- TD 6
  - analyseur LL(1)
- prochain cours
  - analyse syntaxique 2/2