Grammaires et Analyse Syntaxique - Cours 5 Analyse LL(1) dans le cas général

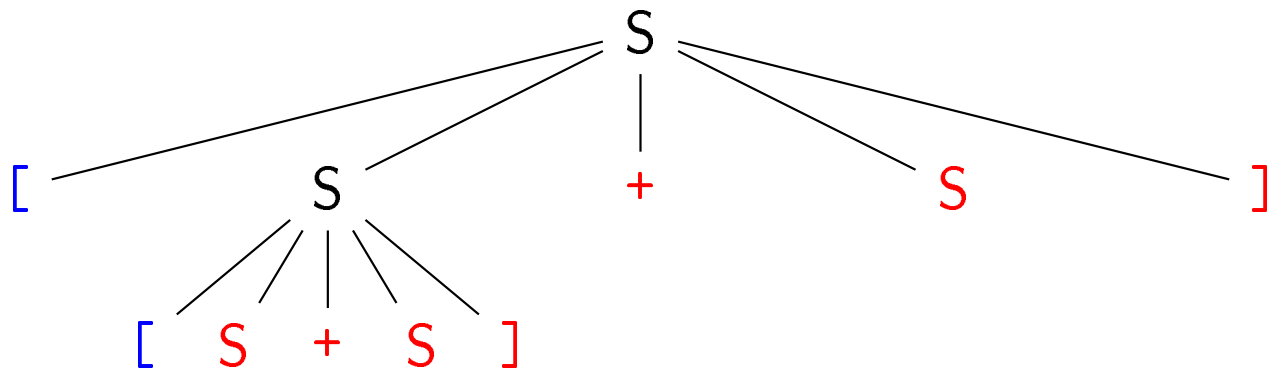
Ralf Treinen

treinen@irif.fr

18 février 2022

Construction d'un arbre de dérivation

Règles de la grammaire : (1) S → `i` (2) S → `[` S `+` S `]`



[	[	i	+	i	]	+	[	i	+	i	]	]

Choisir règle (1) : c'est la seule qui peut produire à partir de S un mot qui commence par `i`.

Grammaires LL(1)

- Intuition derrière les grammaires LL(1) : dans la construction d'une dérivation gauche, le symbole suivant de l'entrée nous indique quelle règle de la grammaire appliquer au non-terminal le plus à gauche de l'arbre de dérivation.
- Conséquence : toute grammaire LL(1) (ou même LL(k)) est non-ambiguë.
- Le premier critère pour être LL(1) s'applique seulement aux grammaires G où tous les côtés droits des règles commencent par un terminal :
  G est LL(1) *ssi* tous les côtés droits des règles pour le *même* non-terminal commencent par des terminaux différents.

Rappel First_1

- Étant donnée une grammaire $G = (\Sigma, N, S, P)$ et un mot $\alpha \in (N \cup \Sigma)^*$

$$\mathrm{First}_1(\alpha) = \{c \in \Sigma \mid \exists w \in \Sigma^*, \alpha \rightarrow^* cw\}$$

- $\mathrm{First}_1(\alpha)$ est l'ensemble des symboles par lesquels un mot terminal dérivé à partir de α peut commencer.

Un meilleur critère pour être LL(1)

- Vu au dernier cours : calcul de First_1 dans le cas où aucun non-terminal est annulant.
- Deuxième critère pour être LL(1), s'applique seulement aux grammaires G où tous les côtés droits des règles sont non vides.
  G est LL(1) *ssi* tous les côtés droits des règles pour le *même* non-terminal ont des ensembles First_1 disjoints.

Exemple vu au dernier cours

- Grammaire $G = (\{\mathsf{F}, \mathsf{S}\}, \{\mathtt{a}, \mathtt{(}, \mathtt{)}, \mathtt{+}\}, \mathsf{S}, P)$ où P est

$$\begin{array}{lcl} \mathsf{F} & \to & \mathtt{a} \\ \mathsf{S} & \to & \mathtt{(}\mathsf{F}\mathtt{+}\mathsf{S}\mathtt{)} \\ \mathsf{S} & \to & \mathsf{F} \end{array}$$

- Le première critère simple ne s'applique pas.
- On obtient pour les côtés droits des règles :

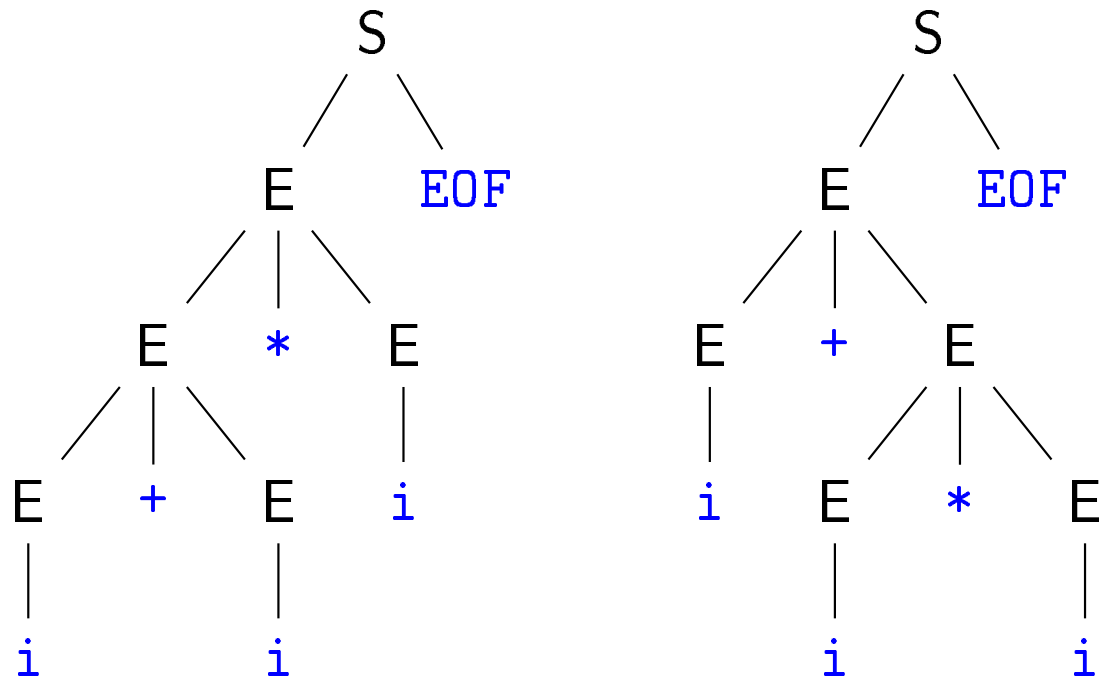
$$\begin{array}{rcl} \mathrm{First}_1(\mathtt{a}) & = & \{\mathtt{a}\} \\ \mathrm{First}_1(\mathtt{(}\mathsf{F}\mathtt{+}\mathsf{S}\mathtt{)}) & = & \{\mathtt{(}\} \\ \mathrm{First}_1(\mathsf{F}) & = & \{\mathtt{a}\} \end{array}$$

Première grammaire pour des expressions arithmétiques

- La grammaire $G_1 = (\{\texttt{i}, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{EOF}\}, \{\mathsf{S}, \mathsf{E}\}, \mathsf{S}, P)$, où P est

$$\mathsf{S} \to \mathsf{E}\ \texttt{EOF} \qquad\qquad \mathsf{E} \to \texttt{i} \mid \mathsf{E}\texttt{+}\mathsf{E} \mid \mathsf{E}\texttt{*}\mathsf{E} \mid \texttt{(}\mathsf{E}\texttt{)}$$

- Cette grammaire décrit les expressions arithmétiques partiellement parenthésées. Elle est ambiguë :



Deuxième grammaire pour les expressions parenthésées

- La grammaire $G_2 = (\{\texttt{i}, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{EOF}\}, \{\mathsf{E}, \mathsf{T}, \mathsf{F}\}, \mathsf{S}, P)$, où P est

$$\begin{array}{lcl} \mathsf{S} & \to & \mathsf{E}\ \texttt{EOF} \\ \mathsf{E} & \to & \mathsf{E}\texttt{+}\mathsf{T} \mid \mathsf{T} \\ \mathsf{T} & \to & \mathsf{T}\texttt{*}\mathsf{F} \mid \mathsf{F} \\ \mathsf{F} & \to & \texttt{(}\mathsf{E}\texttt{)} \mid \texttt{i} \end{array}$$

- Cette grammaire décrit les expressions arithmétiques partiellement parenthésées.

L'exemple des expressions partiellement parenthésées

- Cette grammaire est non-ambiguë ☺
- Intuition : Les "+" peuvent être produits seulement à partir du E. Tout mot engendré par E "au-dessous" d'un * est protégé par des parenthèses (et).
- (Il y a aussi une preuve formelle mais je vous en fais grâce.)
- Exercice : construire les arbres de dérivation pour

```
i+i*i EOF
i+i+i EOF
```

- Cette grammaire est-elle aussi LL(1) ? Ou au moins LL(k) pour un certain $k \in \mathbb{N}$?
- Elle n'est pas LL(k), pour aucun k ! ☹

Un critère pour ne *pas* être LL(k)

Définition

Une grammaire $G = (\Sigma, N, S, P)$ est *récursive à gauche* s'il y a un non-terminal $K \in N$ tel que $K \rightarrow^+ K\alpha$ pour un $\alpha \in (\Sigma \cup N)^*$.

- $\rightarrow^+$: dérivation en *au moins une* étape.
- Exemple : notre grammaire pour les expressions partiellement parenthésées, car E $\rightarrow$ E+T

Définition

- Un non-terminal K est *accessible* s'il existent $\alpha, \beta \in (\Sigma \cup N)^*$ tel que $S \rightarrow^* \alpha K \beta$.
- Un non-terminal K est *productif* s'il existe $w \in \Sigma^*$ tel que K $\rightarrow^*$ w.
- Une grammaire est *réduite* quand tous ses non-terminaux sont accessibles et productifs.

Exemples

- Grammaire $(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}, \mathrm{F}, P)$ avec P :

$$\begin{array}{llllll} \mathrm{A} & \to & \epsilon \mid \mathtt{a} & \mathrm{C} & \to & \mathrm{A\ B} \mid \mathtt{c} \\ \mathrm{B} & \to & \epsilon \mid \mathtt{b} & \mathrm{D} & \to & \mathrm{C\ a\ C} \end{array} \qquad \begin{array}{lll} \mathrm{E} & \to & \mathrm{A\ B} \\ \mathrm{F} & \to & \mathrm{E\ d\ E} \mid \mathrm{A\ F} \end{array}$$

  Accessibles : F, E, A, B. Non-accessibles : C, D.

- Grammaire $(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}, \mathrm{F}, P)$ avec P :

$$\begin{array}{llllll} \mathrm{A} & \to & \epsilon \mid \mathtt{a} & \mathrm{C} & \to & \mathrm{A\ B} \mid \mathtt{c} \\ \mathrm{B} & \to & \mathrm{E\ b} & \mathrm{D} & \to & \mathrm{C\ a\ C} \end{array} \qquad \begin{array}{lll} \mathrm{E} & \to & \mathrm{A\ B} \mid \mathrm{B\ A} \\ \mathrm{F} & \to & \mathrm{E\ d\ E} \mid \mathrm{A\ F} \mid \mathtt{c}\ \mathrm{D} \end{array}$$

  Productif A, C, D, F. Non-productif : B, E.

Récursivité à gauche et LL(1)

Lemme

Si la grammaire G est réduite et récursive à gauche, alors G n'est *pas* LL(k), pour aucun $k \in \mathbb{N}$.

- La raison est :
- On peut faire des dérivations

$$K\gamma \rightarrow^+ K\alpha\gamma \rightarrow^+ K\alpha\alpha\gamma \rightarrow^+ \cdots \rightarrow^+ K\alpha^n\gamma \rightarrow \beta\alpha^n\gamma$$

- Le lookahead ne peut pas nous aider à décider combien de fois appliquer la règle $K \rightarrow K\alpha$ car dans cette dérivation il est toujours le même !

Que faire ?

- On peut transformer la grammaire en une grammaire équivalente (qui définit le même langage), et qui est LL(1).
- Ce n'est pas toujours possible, et il y a deux inconvénients :
  - la grammaire résultante peut être plus grande ;
  - la structure de l'arbre de dérivation peut changer.
- Il y a un troisième inconvénient : la transformation peut introduire des règles $K \rightarrow \epsilon$, il faut donc adapter la technique à ce cas.

La grammaire transformée

- Une grammaire pour les expressions arithmétiques *partiellement* parenthésées.
- Terminaux : $\{\texttt{i}, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{EOF}\}$
grammaire originale	grammaire transformée
$S \to E\ \texttt{EOF}$	$S \to E\ \texttt{EOF}$
$E \to E\texttt{+}T \mid T$	$E \to T\ E'$
$T \to T\texttt{*}F \mid F$	$E' \to \epsilon \mid \texttt{+}E$
$F \to \texttt{(}E\texttt{)} \mid \texttt{i}$	$T \to F\ T'$
	$T' \to \epsilon \mid \texttt{*}T$
	$F \to \texttt{(}E\texttt{)} \mid \texttt{i}$
- Axiome : S

Explication de la transformation

- Les deux règles originales pour le non-terminal E :

$$\begin{aligned} E &\rightarrow T \\ E &\rightarrow E{+}T \end{aligned}$$

- Dans la grammaire d'origine, le non-terminal E engendre une séquence non-vide de non-terminaux T, séparés par des +.
- Dans la grammaire transformée, le non-terminal E′ engendre la suite de cette séquence après un T :

$$\begin{aligned} E &\rightarrow T\ E' \\ E' &\rightarrow \epsilon \mid {+}E \end{aligned}$$

- Pareil pour le non-terminal T.

Non-terminaux annulables

Définition

Soit $G = (\Sigma, N, S, P)$ une grammaire. $K \in N$ est *annulable* si $K \rightarrow^* \epsilon$.

On note souvent EPS l'ensemble des non-terminaux annulables d'une grammaire.

Calcul des non-terminaux annulables

- Si $K \rightarrow \epsilon \in P$ alors K est annulable.
- Si $K \rightarrow K_1 \cdots K_n \in P$, et K_i est annulable pour tout i, alors K est aussi annulable.

Exemple

- Grammaire $(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}, \mathrm{F}, P)$ avec P :

$$\begin{array}{llll} \mathrm{A} \to \epsilon \mid \mathtt{a} & \mathrm{C} \to \mathrm{A\ B} \mid \mathtt{c} & \mathrm{E} \to \mathrm{A\ B\ C} \\ \mathrm{B} \to \epsilon \mid \mathtt{b} & \mathrm{D} \to \mathrm{C\ a\ C} & \mathrm{F} \to \mathrm{E\ d} \mid \mathrm{C\ F} \end{array}$$

- Sont annulables :
  - A, B car il y a ϵ côté droit
  - C car A, B annulables
  - E car A, B, C annulables
- D et F ne sont pas annulables.

Comment calculer First_1 dans le cas général ?

- Imaginez une règle A $\rightarrow$ B C `d` E
- Soit *EPS* l'ensemble des non-terminaux annulant.
- Si B $\notin$ *EPS* : dans cette règle, seulement First_1(B) peut contribuer à First_1(A).
- Si B $\in$ *EPS* : First_1(C) peut aussi contribuer à First_1(A).
- Si B $\in$ *EPS* et C $\in$ *EPS* : `d` doit être dans First_1(A).
- Dans aucun des cas, First_1(E) ne peut contribuer car il se trouve derrière le terminal `d`.

Calcul de First_1 avec non-terminaux annulables

- On fait un graphe, où les nœuds sont les non-terminaux.
- On fait une arête de A vers B quand il y a une règle $B \to K_1 \cdots K_n A\alpha$ où $n \geq 0$, et tous les K_i sont annulables.
- Initialement on met sur un nœud K tous les terminaux a tel qu'il existe une règle $K \to K_1 \cdots K_n a\alpha$ où tous les K_i sont annulables.
- Puis on propage les valeurs dans le sens des flèches.

Calcul de First_1 sur l'exemple

$$
\begin{array}{llll}
E & \to & T\ E' & E' \to \epsilon \mid \texttt{+}E \\
T & \to & F\ T' & T' \to \epsilon \mid \texttt{*}T \\
F & \to & \texttt{(}E\texttt{)} \mid \texttt{i} & S \to E\ \texttt{EOF}
\end{array}
$$

- Non-terminaux annulables : E', T'
- Initialisation :

$$
\begin{array}{ccccccccc}
\{\texttt{i},\texttt{(}\} & & \{\} & & \{\} & & \{\} & \{\texttt{*}\} & \{\texttt{+}\} \\
F & \longrightarrow & T & \longrightarrow & E & \longrightarrow & S & T' & E'
\end{array}
$$

- Propagation :

$$
\begin{array}{ccccccccc}
\{\texttt{i},\texttt{(}\} & & \{\texttt{i},\texttt{(}\} & & \{\texttt{i},\texttt{(}\} & & \{\texttt{i},\texttt{(}\} & \{\texttt{*}\} & \{\texttt{+}\} \\
F & \longrightarrow & T & \longrightarrow & E & \longrightarrow & S & T' & E'
\end{array}
$$

Calcul de $\mathrm{First}_{\leq 1}$ dans le cas général

- On calcule maintenant pour les côtés droits de la grammaire $\mathrm{First}_{\leq 1}(\alpha) := \{w : 1 \mid \alpha \rightarrow^* w, w \in \Sigma^*\}$
- La différence avec $\mathrm{First}_1(\alpha)$ est que $\mathrm{First}_{\leq 1}(\alpha)$ peut aussi contenir ϵ.
- $\mathrm{First}_{\leq 1}(\epsilon) = \{\epsilon\}$
- pour tout $a \in \Sigma$: $\mathrm{First}_{\leq 1}(a\alpha) = \{a\}$
- pour tout $A \in N$:

$$\mathrm{First}_{\leq 1}(A\alpha) = \begin{cases} \mathrm{First}_1(A) & \text{si } A \notin EPS \\ \mathrm{First}_1(A) \cup \mathrm{First}_{\leq 1}(\alpha) & \text{si } A \in EPS \end{cases}$$

Calcul de $\mathrm{First}_{\leq 1}$ des côtés droits dans l'exemple

$$
\begin{array}{lcllcl}
\mathsf{E} & \to & \mathsf{T\ E'} & \mathsf{E'} & \to & \epsilon \mid \texttt{+}\mathsf{E} \\
\mathsf{T} & \to & \mathsf{F\ T'} & \mathsf{T'} & \to & \epsilon \mid \texttt{*}\mathsf{T} \\
\mathsf{F} & \to & \texttt{(}\mathsf{E}\texttt{)} \mid \texttt{i} & \mathsf{S} & \to & \mathsf{E\ EOF}
\end{array}
$$

A	$\mathrm{First}_1(A)$
S	{i, (}
E	{i, (}
E'	{+}
T	{i, (}
T'	{*}
F	{i, (}

$EPS = \{\mathsf{E'}, \mathsf{T'}\}$

α	$\mathrm{First}_{\leq 1}(\alpha)$
T E'	{i, (}
ϵ	$\{\epsilon\}$
+ E	{+}
F T'	{i, (}
* T	{*}
(E)	{(}
i	{i}
E EOF	{i, (}

Nous avons besoin de plus d'information !

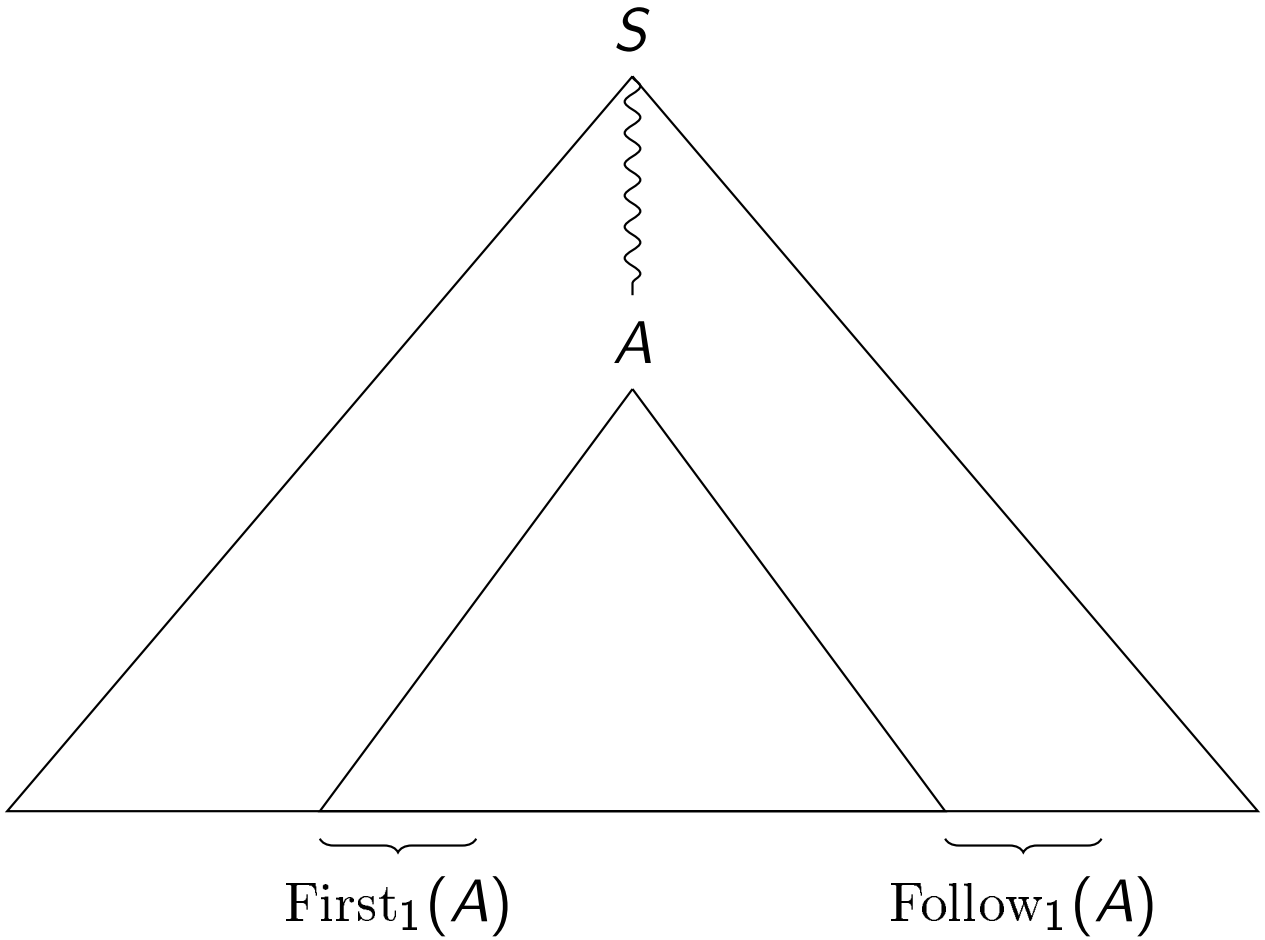
- Le calcul de $\mathrm{First}_{\leq 1}$ n'est plus suffisant pour savoir quelle production appliquer !
- Exemple : E′ $\rightarrow \epsilon \mid$ +E

α	$\mathrm{First}_{\leq 1}(\alpha)$
ϵ	$\{\epsilon\}$
+ E	$\{+\}$

Si nous voyons + alors il faut utiliser la deuxième alternative pour réécrire E′. Mais quand faut-il appliquer la première ?

- Il nous manque une information : quels sont les symboles terminaux qui peuvent *suivre* un mot produit par un non-terminal ?

First_1 et Follow_1



La fonction Follow_1

Définition

Soit $G = (\Sigma, N, S, P)$ une grammaire. La fonction $\mathrm{Follow}_1 : N \to 2^{\Sigma}$ est définie par

$$\mathrm{Follow}_1(A) = \{c \mid S \to^* \beta A \gamma, \beta, \gamma \in (N \cup \Sigma)^*, c \in \mathrm{First}_1(\gamma)\}$$

Explication

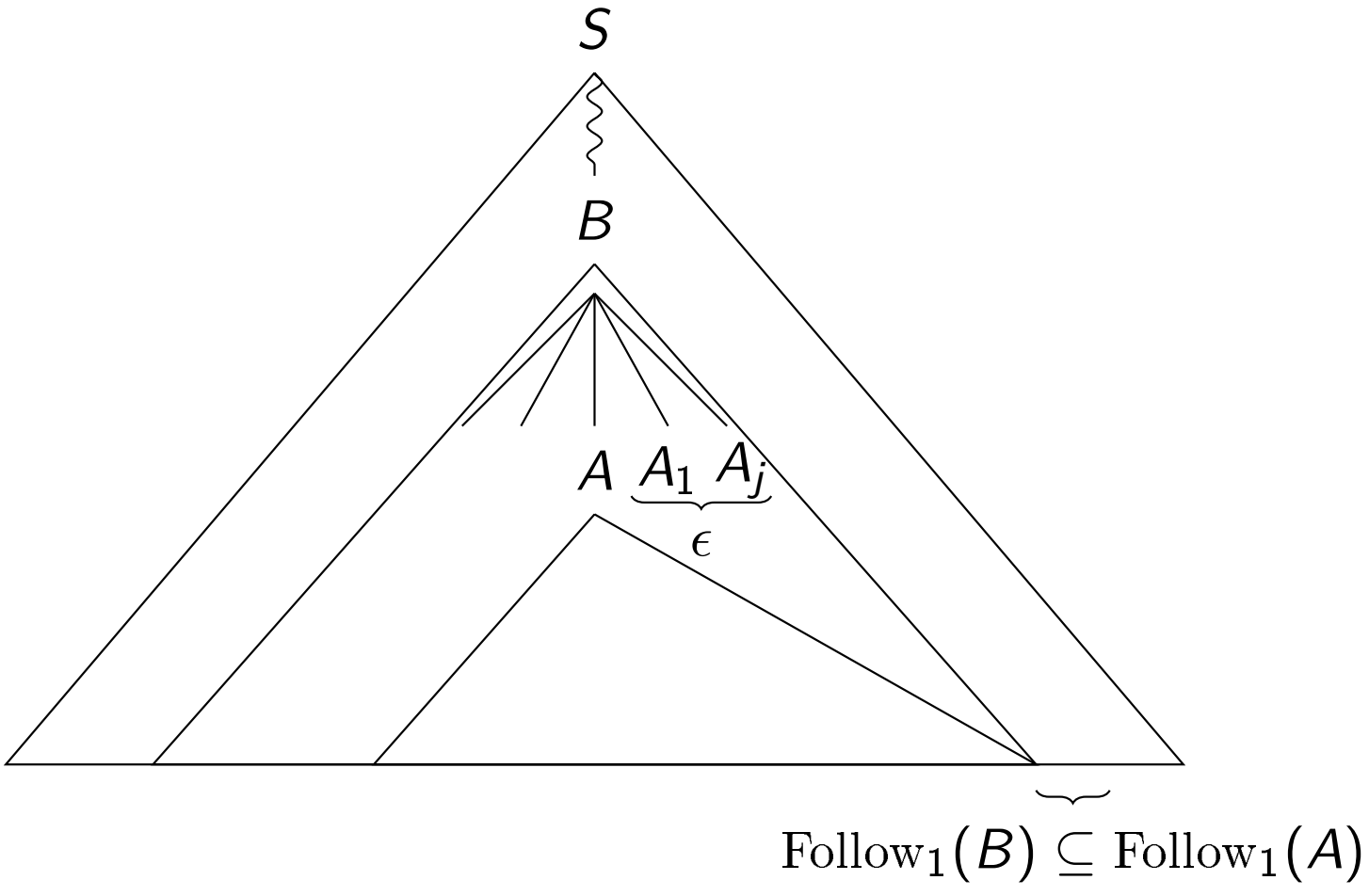
$\mathrm{Follow}_1(A)$ est l'ensemble de tous les symboles terminaux qui peuvent, dans des mots de $\mathcal{L}(G)$, suivre un mot dérivé de A.

Calcul de Follow_1 pour les non-terminaux

- Grammaire $G = (\Sigma, N, S, P)$.
- On fait un graphe, avec N comme ensemble de nœuds.
- On fait une arête de A vers B quand il y une règle de la forme $A \to \alpha B B_1 \dots B_n$ où $B_1, \dots B_n \in EPS$.
- On ajoute des symboles de Σ comme valeurs aux nœuds. Initialement, on ajoute à un nœud A, pour toutes les règles $\dots \to \dots A\alpha$, l'ensemble $\mathrm{First}_1(\alpha)$.
- Puis on propage les valeurs dans le sens des flèches, jusqu'à ne plus pouvoir propager.

Follow_1 : propagation de gauche à droite

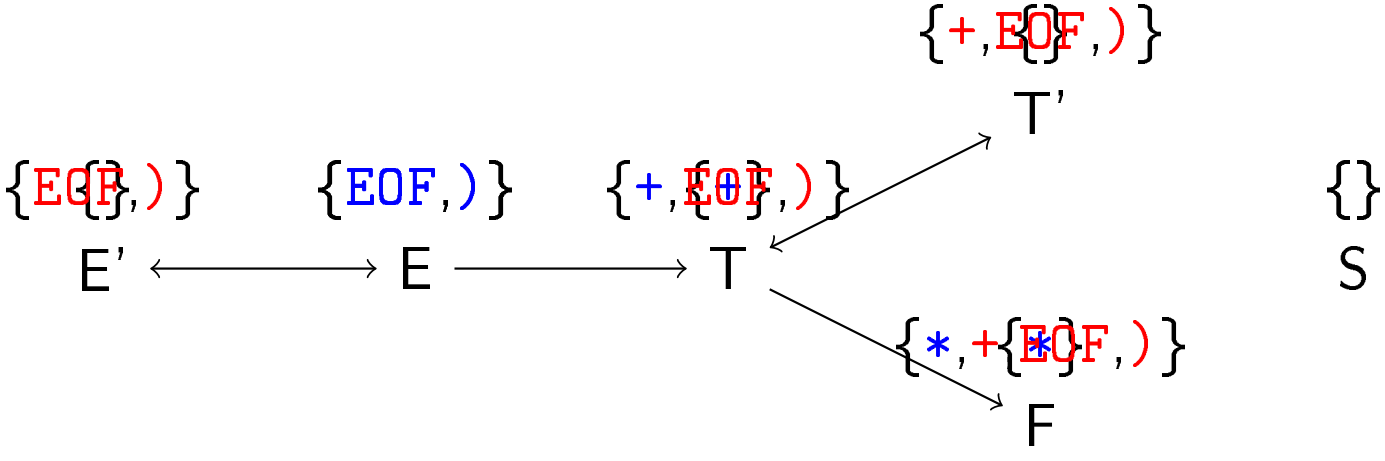
Cas d'une règle $B \to \ldots AA_1 \ldots A_j$ avec $A_1, \ldots, A_j \in EPS$



Calcul de Follow_1 sur l'exemple

$$\begin{array}{lllllllll} \mathsf{E} & \to & \mathsf{T}\ \mathsf{E}' & \mathsf{E}' & \to & \epsilon \mid \texttt{+}\mathsf{E} & \mathsf{T} & \to & \mathsf{F}\ \mathsf{T}' \\ \mathsf{T}' & \to & \epsilon \mid \texttt{*}\mathsf{T} & \mathsf{F} & \to & \texttt{(}\mathsf{E}\texttt{)} \mid \texttt{i} & \mathsf{S} & \to & \mathsf{E}\ \texttt{EOF} \end{array}$$

- $EPS = \{\mathsf{E}', \mathsf{T}'\}$ $\quad \mathrm{First}_1(\mathsf{E}') = \{\texttt{+}\}$ $\quad \mathrm{First}_1(\mathsf{T}') = \{\texttt{*}\}$
- Initialisation (en bleu)



- Propagation : ajouter les symboles rouges

Le critère général pour être LL(1)

Théorème

La grammaire $G = (\Sigma, N, S, P)$ est LL(1) ssi pour toutes les alternatives $A \to \alpha_1 \mid \ldots \mid \alpha_n$:

1. $\mathrm{First}_{\leq 1}(\alpha_1), \ldots, \mathrm{First}_{\leq 1}(\alpha_n)$ sont disjoints entre eux ;
2. Si $\epsilon \in \mathrm{First}_{\leq 1}(\alpha_i)$, alors pour tous $j \neq i$:

$$\mathrm{First}_{\leq 1}(\alpha_j) \cap \mathrm{Follow}_1(A) = \emptyset$$

Remarque

Condition (1) implique qu'au plus un des ensembles $\mathrm{First}_{\leq 1}(\alpha_i)$ contient ϵ.

Le critère sur l'exemple

Non-terminal	Cas 1	$\text{First}_{\leq 1}$	Cas 2	$\text{First}_{\leq 1}$
E	T E′	{i, (}		
E′	ϵ	{ϵ}	+ E	{+}
T	F T′	{i, (}		
T′	ϵ	{ϵ}	* T	{*}
F	(E)	{(}	i	{i}
S	E EOF	{i, (}		

- $\text{Follow}_1(\text{E}') = \{), \texttt{EOF}\}$ disjoint avec {+} ☺
- $\text{Follow}_1(\text{T}') = \{\texttt{EOF},), +\}$ disjoint avec {*} ☺
- {(} disjoint avec {i} ☺
- Conclusion : la grammaire est LL(1) !

Comment choisir la règle dans l'analyse syntaxique

Soit $A \to \alpha_1 \mid \ldots \mid \alpha_n$ une alternative. Il y a deux cas :

1. Soit aucun des $\mathrm{First}_{\leq 1}(\alpha_i)$ ne contient ϵ :
   comme avant :
   - On choisit la règle $A \to \alpha_i$ quand le symbole suivant est dans $\mathrm{First}_{\leq 1}(\alpha_i)$ (ils sont tous disjoints).
   - Erreur si aucun tel i existe
2. Soit il existe un (seul) α_i avec $\epsilon \in \mathrm{First}_{\leq 1}(\alpha_i)$:
   - si le symbole suivant est dans $\mathrm{First}_{\leq 1}(\alpha_j)$: choisir $A \to \alpha_j$, pour $1 \leq j \leq n$.
   - si le symbole suivant est dans $\mathrm{Follow}_1(A)$: choisir $A \to \alpha_i$.
   - sinon Erreur.

Fichier parser.ml I

```
open Tree
open Reader

exception Error of string
let rec parse_S () =
  match lookahead () with
  | Ch 'i' | Ch '(' -> begin (* S -> E *)
      let x = parse_E () in
      Node("S",[x])
    end
  | _ -> raise (Error "parsing␣S")
and parse_E () =
  match lookahead () with
  | Ch 'i' | Ch '(' -> begin (* E -> T E' *)
      let x1 = parse_T () in
      let x2 = parse_Eprime () in
      Node("E",[x1;x2])
    end
```

Fichier parser.ml II

```
  | _ -> raise (Error "parsing E")
and parse_Eprime () =
  match lookahead () with
  | Ch ')' | EOF -> (* E' -> epsilon *)
    Node("E'",[Epsilon])
  | Ch '+' -> begin (* E' -> + E *)
      eat (Ch '+');
      let x = parse_E () in
      Node("E'",[Leaf '+';x])
    end
  | _ -> raise (Error "parsing E'")
and parse_T () =
  match lookahead () with
  | Ch 'i' | Ch '(' -> begin (* T -> F T' *)
      let x1 = parse_F () in
      let x2 = parse_Tprime () in
      Node("T",[x1;x2])
    end
```

Fichier parser.ml III

```
  | _ -> raise (Error "parsing T")
and parse_Tprime () =
  match lookahead () with
  | Ch ')' | Ch '+' | EOF -> (* T' -> epsilon *)
     Node("T'",[Epsilon])
  | Ch '*' -> begin (* T' -> * T *)
      eat (Ch '*');
      let x = parse_T () in
      Node("T'",[Leaf '*';x])
    end
  | _ -> raise (Error "parsing E'")
and parse_F () =
  match lookahead () with
  | Ch '(' -> begin (* F -> ( E ) *)
      eat (Ch '(');
      let x = parse_E () in
      eat (Ch ')');
      Node("F",[Leaf '(';x;Leaf ')'])
```

Fichier parser.ml IV

```
    end
  | Ch 'i' -> begin (* F -> i *)
      eat (Ch 'i');
      Node("F",[Leaf 'i'])
    end
  | _ -> raise (Error "parsing F")

let parse () = parse_S ()
```