Grammaires et Analyse Syntaxique - Cours 4
Introduction à l'analyse LL(1)

Ralf Treinen

INFORMATIQUE Sciences Université de Paris

INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

treinen@irif.fr

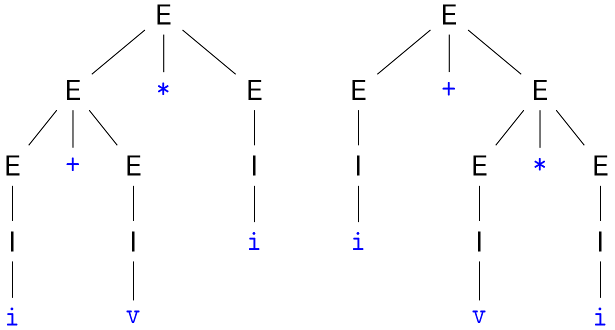
18 février 2022

Vue la semaine dernière

- Grammaires
- Dérivations (gauches, droites)
- Arbres de dérivation
- "*dérivation* = *arbre de dérivation* + *stratégie* "
- Une grammaire G est ambiguë quand il existe un mot w qui a deux arbres de dérivation différents
  - équivalent : qui a deux dérivations gauches différentes
  - équivalent : qui a deux dérivations droites différentes

Deux arbres de dérivation de i+v*i dans G_1

$$E \rightarrow E+E \mid E*E \mid (E) \mid I \qquad I \rightarrow i \mid v$$



Écartez les grammaires ambiguës

- Les grammaires ambiguës sont (en principe) interdites pour l'analyse grammaticale.
- Raison : Nous ne voulons pas seulement savoir si un mot est accepté par la grammaire ou non, mais aussi connaître son arbre de dérivation, qui va être utilisé dans la suite.
- C'est une différence avec le cours AAL3, où on s'est seulement intéressé à l'acceptation d'un mot (par un automate, une regexp)
- "En Principe" : certains outils (voir plus tard) acceptent des grammaires ambiguës, mais seulement avec une spécification supplémentaire qui permet de désambiguïser.

Plusieurs techniques pour l'analyse grammaticale

- *Analyse descendante* : construction de l'arbre de dérivation, à partir de l'axiome jusqu'aux feuilles.
  Ordre de construction : parcours préfixe de l'arbre.
  C'est l'approche que nous étudions aujourd'hui et la semaine prochaine.
- *Analyse ascendante* : construction d'un arbre de dérivation à partir des feuilles jusqu'à l'axiome. Plus complexe à maîtriser, mais aussi plus puissante.
  C'est l'approche que nous commencerons à étudier dans deux semaines.

Construction d'un arbre de dérivation

- Dans la construction d'un arbre de dérivation (ou, d'une dérivation), il y a à chaque moment deux choix à faire :
  - le non-terminal qu'on va remplacer à l'aide d'une règle de la grammaire,
  - une fois le non-terminal choisi, la règle parmi celles qui ont ce non-terminal sur le côté gauche.
- Nous avons vu la semaine dernière que le premier choix n'est pas essentiel : on peut imposer une stratégie pour choisir le non-terminal à remplacer (par ex., celui qui est le plus à gauche).

Exploration complète de l'espace de recherche ?

- Une façon de réaliser une analyse grammaticale est maintenant d'essayer simplement toutes les possibilités de choisir des règles.
- Cela donne lieu à un algorithme *non-déterministe* :
  - soit par *retour en arrière* (angl. : backtracking)
  - soit par *programmation dynamique*
- Approche complète : on est sûr de trouver un arbre de dérivation si le mot est dans le langage ☺
- Problème : efficacité ☹
- On cherche des solutions efficaces, éventuellement en imposant des restrictions aux grammaires qu'on peut traiter.

Quelle efficacité cherche-t-on ?

- Le temps d'exécution d'un analyseur grammaticale est au moins linéaire dans la longueur du texte à analyser (c-à-d la longueur du flot des tokens). Évidemment on ne peut pas faire mieux.
- Puisqu'on cherche aussi à construire l'arbre de dérivation, on ne peut même pas faire mieux que la taille de l'arbre de dérivation.
- La taille de l'arbre de dérivation est $\geq$ la taille de l'entrée (car chaque symbole de l'entrée est une feuille de l'arbre).
- On veut aussi que l'analyseur fasse un seul passage sur le texte.

Comment obtenir une solution efficace ?

- Il faut maîtriser le choix de la règle de la grammaire par laquelle on va remplacer un non-terminal.
- On ne peut pas demander qu'il y ait une seule règle par non-terminal (car dans ce cas la grammaire est complètement triviale).
- Sur quoi baser le choix de la règle ?
- Sur la suite du mot pour lequel on cherche à construire l'arbre de dérivation !

Exemple

- Grammaire $G = (\Sigma, N, S, P)$ où
- $\Sigma = \{\texttt{i}, \texttt{+}, \texttt{[}, \texttt{]}\}$
- $N = \{\mathsf{S}\}$
- $S = \mathsf{S}$
- P consiste en les règles suivantes :

$$\begin{aligned} \mathsf{S} &\rightarrow \texttt{i} \qquad (1) \\ \mathsf{S} &\rightarrow \texttt{[S+S]} \qquad (2) \end{aligned}$$

- $\mathcal{L}(G)$: expressions complètement parenthésées, construites avec la constante $\texttt{i}$ et l'opérateur binaire $\texttt{+}$.

Construction d'un arbre de dérivation (1)

Productions : (1) $S \to$ i (2) $S \to$ [S+S]

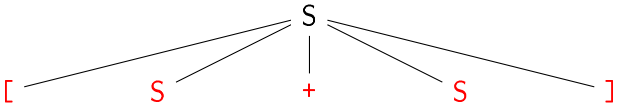
S

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

Choisir règle (2) : c'est la seule qui peut produire à partir de S un mot qui commence par [.

Construction d'un arbre de dérivation (2)

Productions : (1) S $\rightarrow$ i (2) S $\rightarrow$ [S+S]

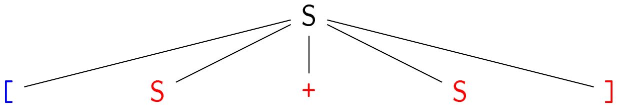


[	[	i	+	i	]	+	[	i	+	i	]	]

Le premier non-terminal du mot des feuilles est [.

Construction d'un arbre de dérivation (3)

Productions : (1) S → i (2) S → [S+S]

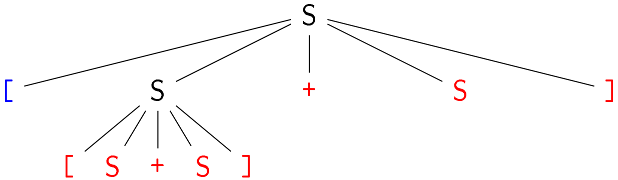


[	[	i	+	i	]	+	[	i	+	i	]	]

Choisir règle (2) : c'est la seule qui peut produire à partir de S un mot qui commence par [.

Construction d'un arbre de dérivation (4)

Productions : (1) S → i (2) S → [S+S]

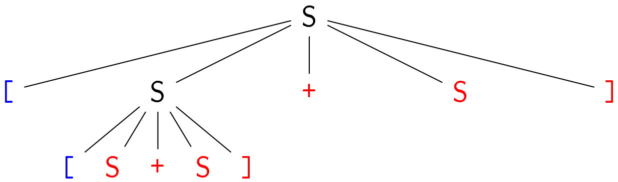


[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est [.

Construction d'un arbre de dérivation (5)

Productions : (1) S → `i` (2) S → `[S+S]`

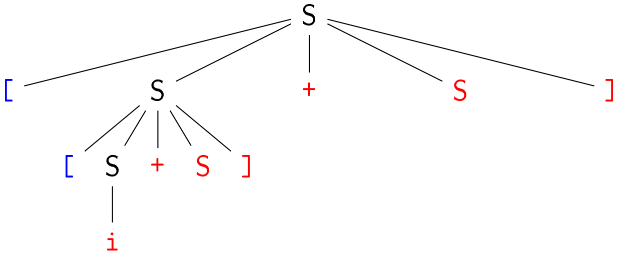


[	[	i	+	i	]	+	[	i	+	i	]	]

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

Construction d'un arbre de dérivation (6)

Productions : (1) S → i (2) S → [S+S]

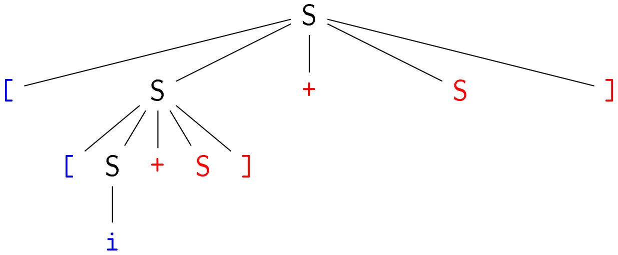


[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est i.

Construction d'un arbre de dérivation (7)

Productions : (1) S → i (2) S → [S+S]

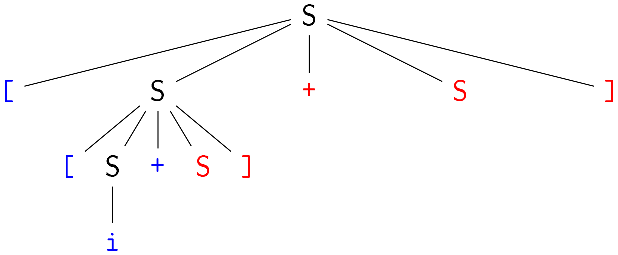


[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est +.

Construction d'un arbre de dérivation (8)

Productions : (1) S → `i` (2) S → `[S+S]`

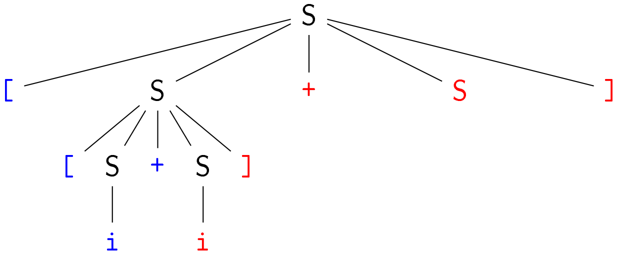


[	[	i	+	i	]	+	[	i	+	i	]	]

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

Construction d'un arbre de dérivation (9)

Productions : (1) S → i (2) S → [S+S]

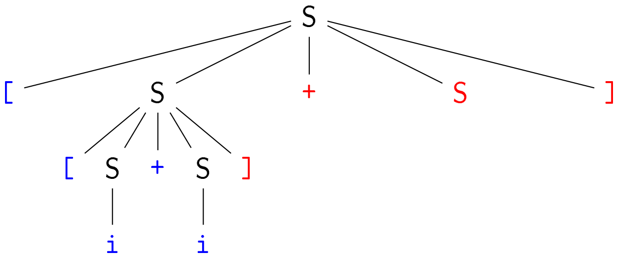


[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est i.

Construction d'un arbre de dérivation (10)

Productions : (1) S $\rightarrow$ i (2) S $\rightarrow$ [S+S]

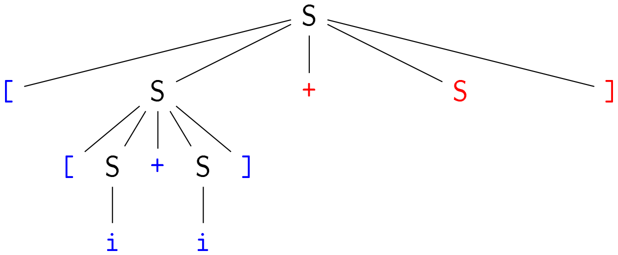


[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est].

Construction d'un arbre de dérivation (11)

Productions : (1) S → i (2) S → [S+S]

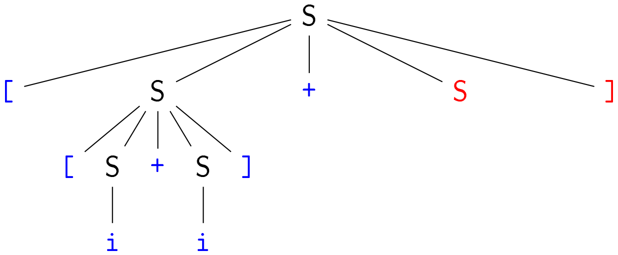


[	[	i	+	i	]	+	[	i	+	i	]	]

Seule la règle (2) peut produire à partir de S un mot qui commence par +.

Construction d'un arbre de dérivation (12)

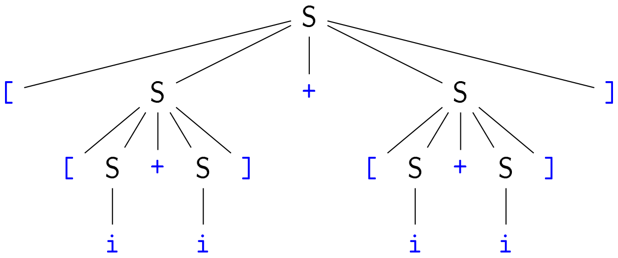
Productions : (1) S → i (2) S → [S+S]



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

Choisir règle (2) : c'est la seule qui peut produire à partir de S un mot qui commence par [.

etc. etc.



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

Construction terminée !

Ce qu'on a vu sur l'exemple :

- Il y a deux types d'actions :
  - consommer en parallèle un terminal du préfixe du mot des feuilles déjà construit, et le même symbole de l'entrée ;
  - ajouter des fils à une feuille de l'arbre de dérivation partiel.
- Pour choisir la règle de la grammaire, on regarde en avant quel est le symbole suivant de l'entrée que nous aurions à consommer (lookahead).

Grammaires LL(1)

En fait, l'algorithme que nous avons vu sur l'exemple appartient à la classe *LL(1)* :

- le premier *L* indique qu'on parcourt l'entrée de la gauche (angl. : left) à la droite ;
- le deuxième *L* indique qu'on construit une dérivation gauche (angl. : left), c.-à-d. un arbre de dérivation dans un ordre préfixe ;
- le nombre *1* indique que nous utilisons la connaissance de 1 caractère dans la partie de l'entrée qui reste à consommer, pour déterminer la règle à appliquer (lookahead=1).

Grammaires LL(k)

- Idée : on peut déterminer la règle de production à appliquer au non-terminal le plus à gauche de l'arbre de dérivation en regardant les k symboles suivants de l'entrée (lookahead=k)
- Définition précise à venir.
- Généralisation des LL(1).
- En pratique ce sont surtout les grammaires LL(1) qui nous intéressent.

Notation : $w : k$

Définition

Soit $w \in \Sigma^*$ un mot, et $k \in \mathbb{N}$. On définit

- si $|w| \leq k$ alors $w : k = w$
- si $|w| > k$ alors $w : k = x$ tel que $w = xy$ et $|x| = k$

Explication

- $w : k$ est le préfixe de longueur k du mot w, ou le mot w entier si la longueur de w est inférieure à k.
- $abcdefg : 3 = abc$
- $abcd : 7 = abcd$

Définition LL(k)

Définition

Soit $G = (\Sigma, N, S, P)$ une grammaire algébrique, $k \in \mathbb{N}$. G est dite *LL(k)* ssi

- S'il existe deux dérivations *gauches*

$$S \to^* uY\alpha \to u\beta\alpha \to^* ux$$
$$S \to^* uY\alpha \to u\gamma\alpha \to^* uy$$

  où $Y \in N$; $u, x, y \in \Sigma^*$; $\alpha \in (N \cup \Sigma)^*$; $Y \to \beta, Y \to \gamma \in P$.
- si $\beta \neq \gamma$
- alors $x : k \neq y : k$

Explication de la définition de LL(k)

- On a déjà consommé u, partie initiale du mot d'entrée.
- Le non-terminal le plus à gauche à réécrire est maintenant Y.
- Dans les deux cas considérés, le mot d'entrée continue une fois par le mot x, l'autre fois par le mot y.
- En regardant les k premiers caractères de la suite du mot d'entrée, on peut maintenant décider comment réécrire le non-terminal Y.

Conséquences

- Toute grammaire G qui est LL(k) est *non-ambiguë* : tout mot du langage $\mathcal{L}(G)$ a une seule dérivation gauche, et donc un seul arbre de dérivation.
- Il existe un algorithme efficace pour la construction de cet arbre de dérivation.
- Question : comment savoir si une grammaire est LL(k) ?
- Notre définition parle de dérivations quelconques, ce n'est pas une méthode de décision efficace. Comment le décider efficacement en regardant la grammaire ?

Un premier critère simple pour être LL(1)

Lemme

Soit G une grammaire où tous les côtés droits de règles commencent par un terminal.
G est LL(1) ssi pour tout non-terminal, les côtés droits de toutes les règles pour ce non-terminal commencent par des terminaux différents.

Exemple

La grammaire de l'exemple précédent :

$$\begin{array}{lcl} S & \rightarrow & \mathtt{i} \\ S & \rightarrow & \mathtt{[S+S]} \end{array}$$

satisfait le critère, et est donc LL(1).

Exemple d'une grammaire qui n'est *pas* LL(1)

- Grammaire $G_1 = (\Sigma, N, S, P)$ où
- $\Sigma = \{\texttt{i}, \texttt{+}, \texttt{*}, \texttt{[}, \texttt{]}\}$
- $N = \{\mathsf{S}\}$
- $S = \mathsf{S}$
- P consiste en les règles suivantes :

$$
\begin{aligned}
\mathsf{S} &\rightarrow \texttt{i} && (3)\\
\mathsf{S} &\rightarrow \texttt{[}\mathsf{S}\texttt{+}\mathsf{S}\texttt{]} && (4)\\
\mathsf{S} &\rightarrow \texttt{[}\mathsf{S}\texttt{*}\mathsf{S}\texttt{]} && (5)
\end{aligned}
$$

- Pourquoi n'est-elle pas LL(1) ?
- Peut-on la transformer en une grammaire LL(1) ?

Transformation en une grammaire LL(1)

- Si une grammaire n'est pas LL(1) c'est souvent qu'on a à choisir entre deux règles, mais on n'a pas encore suffisament d'informations pour faire ce choix.
- Solution : Retardez le choix !
- par exemple, avec un non-terminal supplémentaire O :

$$S \rightarrow \texttt{i} \tag{6}$$
$$S \rightarrow \texttt{[}\ S\ O\ S\ \texttt{]} \tag{7}$$
$$O \rightarrow \texttt{+} \tag{8}$$
$$O \rightarrow \texttt{*} \tag{9}$$

Vers le deuxième critère pour être LL(1)

- Problème : Le premier critère est trop restrictif car il ne permet pas des règles où le côté droit commence par un non-terminal :

$$A \rightarrow BA \mid B \tag{10}$$
$$B \rightarrow \ldots \tag{11}$$

- On a besoin d'un critère pour être LL(1) qui marche aussi en présence de règles de production où le côté droit commence par un non-terminal.

La fonction $\mathrm{First}_{\leq k}$

Définition

Soit $G = (\Sigma, N, S, P)$ une grammaire, et $k \in \mathbb{N}$. Nous définissons une fonction

$$\mathrm{First}_{\leq k} : (N \cup \Sigma)^* \to 2^{\Sigma^*}$$

par

$$\mathrm{First}_{\leq k}(\alpha) = \{w : k \mid w \in \Sigma^*, \alpha \to^* w\}$$

Explication

$\mathrm{First}_{\leq k}(\alpha)$ est l'ensemble des préfixes de longueur k des mots terminaux qu'on peut obtenir à partir de α.

La fonction First_1

- On pratique c'est le cas $k = 1$ qui nous intéresse : c'est suffisant, et plus simple à traiter que le cas d'un k général.
- $\mathrm{First}_1 : (N \cup \Sigma)^* \to 2^{\Sigma}$
- $\mathrm{First}_1(\alpha) = \{c \in \Sigma \mid \exists w \in \Sigma^* : \alpha \to^* cw\}$
- On a donc que $\mathrm{First}_1(\alpha) = \mathrm{First}_{\leq 1}(\alpha) - \{\epsilon\}$
- Cette semaine nous sommes sous l'hypothèse qu'il n'y a pas de productions de la forme $\mathrm{N} \to \epsilon$, donc on a même $\mathrm{First}_1(N) = \mathrm{First}_{\leq 1}(N)$, *mais ca changera dans le cas général*.

Un meilleur critère pour être LL(1)

Lemme

Soit G une grammaire *sans productions de la forme $N \to \epsilon$*. G est LL(1) si est seulement si pour toutes règles différentes :

$$\begin{array}{lcl} \mathsf{N} & \to & \alpha \\ \mathsf{N} & \to & \beta \end{array}$$

on a que $\mathrm{First}_1(\alpha) \cap \mathrm{First}_1(\beta) = \emptyset$.

Exemple

Toujours sur le même exemple :

$$\begin{array}{rcl} \mathrm{First}_1(\mathtt{i}) & = & \{\mathtt{i}\} \\ \mathrm{First}_1(\mathtt{[S+S]}) & = & \{\mathtt{[}\} \end{array}$$

Calcul de First_1 pour les non-terminaux

- Grammaire $G = (\Sigma, N, S, P)$. *Hypothèse : aucune production $N \to \epsilon$.*
- On fait un graphe, avec N comme ensemble de nœuds.
- On fait une arête de A vers B quand il y a dans P une production de la forme B $\to$ Aα.
- On ajoute des symboles de Σ comme valeurs aux nœuds. Intialement, on ajoute à un nœud A la valeur a quand il y a dans P une production A $\to$ aα.
- Puis on propage les valeurs dans le sens des flèches, jusqu'à ce qu'on ne puisse plus rien propager.

Exemple

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

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

- Initialisation : $\overset{\{\mathtt{a}\}}{\mathsf{F}} \longrightarrow \overset{\{\mathtt{(}\}}{\mathsf{S}}$
- Propagation : $\overset{\{\mathtt{a}\}}{\mathsf{F}} \longrightarrow \overset{\{\mathtt{(},\mathtt{a}\}}{\mathsf{S}}$
- Résultat : $\mathrm{First}_1(\mathsf{F}) = \{\mathtt{a}\}$, $\mathrm{First}_1(\mathsf{S}) = \{\mathtt{(}, \mathtt{a}\}$.

Calcul de First_1 sur des séquences de symboles

- Toujours sous l'hypothèse qu'on n'a pas de règle N $\rightarrow \epsilon$
- On étend la fonction First_1 à des séquences *non vides* de symboles :

$$\mathrm{First}_1(x\alpha) = \begin{cases} \{x\} & \text{si } x \in \Sigma \\ \mathrm{First}_1(x) & \text{si } x \in N \end{cases}$$

- Nous allons utiliser cette généralisation pour calculer les First_1 des côtés droits des règles de la grammaire.

Exemple (2)

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

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

- On obtient donc :

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

Reconnaître la fin de l'entrée

- Avant de faire l'implémentation, il reste un petit problème : il faut s'assurer que l'entrée entière est un mot accepté par la grammaire.
- Solution :
  - l'analyse lexicale envoie un jeton qui signale la fin de l'entrée (par exemple, EOF pour *end of file*)
  - remplacer l'axiome par S′, avec une règle

    $$S' \to S\ \texttt{EOF}$$

    où S est l'ancien axiome de la grammaire.

Le même exemple avec reconnaissance de la fin

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

$$
\begin{array}{rcl}
\mathsf{F} & \rightarrow & \mathtt{a} \\
\mathsf{S} & \rightarrow & \mathtt{(}\mathsf{F}\mathtt{+}\mathsf{S}\mathtt{)} \\
\mathsf{S} & \rightarrow & \mathsf{F} \\
\mathsf{S}' & \rightarrow & \mathsf{S}\ \mathtt{EOF}
\end{array}
$$

- On obtient pour les côtés droits des règles :

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

Implémentation en OCaml

- L'analyseur grammatical (parser) consiste en plusieurs fonctions mutuellement récursives :
  - Une fonction par non-terminal
  - Distinction de cas, un cas par règle, plus un cas d'erreur
- Un module pour demander des jetons de l'entrée, avec
  - une fonction `lookahead` pour obtenir un jeton sans avancer dans l'entrée,
  - un fonction `eat` qui consomme un jeton, et qui avance d'un cran dans l'entrée.

Fichier reader.mli I

```
(* module for a lookahead(1) reader from standard input*)

exception Error of string

(* a token is a character, or the end-of-file marker *)
type token = Ch of char | EOF

(* return the next token, do not advance the read pointer *)
val lookahead : unit -> token

(* [(eat t)] advances the read pointer if the next *)
(* token is t and throws an Error otherwise. *)
val eat : token -> unit
```

Fichier tree.mli I

```
(* module of derivation trees *)

(* type of trees. Inner nodes are labeled with strings, *)
(* leaves are labeled with chars. *)
(* Inner nodes may have an arbitrary number of children. *)
type t =
  | Node of string * t list
  | Leaf of char

(* print tree to stdout *)
val print: t -> unit
```

Fichier parser.ml I

```
open Tree
open Reader

exception Error of string
let rec parse_F () =
  match lookahead () with
  | Ch 'a' -> begin (* F -> a *)
      eat (Ch 'a');
      Node("F",[Leaf 'a'])
    end
  | _ -> raise (Error "parsing F")
and parse_S () =
  match lookahead () with
  | Ch 'a' -> begin (* S -> F *)
      let x = parse_F () in
      Node("S",[x])
    end
  | Ch '(' -> begin (* S -> (F+S) *)
```

Fichier parser.ml II

```
        eat (Ch '(');
        let x1 = parse_F () in
        eat (Ch '+');
        let x2 = parse_S () in
        eat (Ch ')');
        Node("S",[Leaf '(';x1;Leaf '+'; x2; Leaf ')'])
      end
    | _ -> raise (Error "parsing␣S")
  and parse_Sprime () =
    match lookahead () with
    | Ch 'a' | Ch '(' -> begin (* S' -> S EOF *)
        let x = parse_S () in
        eat EOF;
        Node("Sprime",[x; Leaf '#'])
      end
    | _ -> raise (Error "parsing␣Sprime")

let parse () = parse_Sprime ()
```