

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

Ralf Treinen



treinen@irif.fr

13 février 2025

© Ralf Treinen 2020–2025

Petit rappel des grammaires

- ▶ Exemple d'une grammaire algébrique :

$$(\{a, b\}, \{A\}, A, A \rightarrow aAb \mid \epsilon)$$

- ▶ Exemple d'une dérivation dans cette grammaire

$$A \rightarrow aAb \rightarrow aaAbb \rightarrow aaaAbbb \rightarrow aaaAbbb \rightarrow aaabbb$$

- ▶ Langage engendré par cette grammaire :

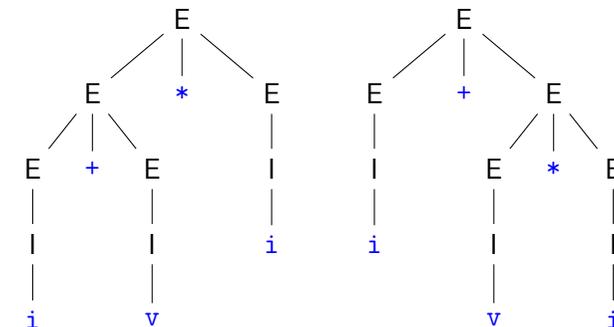
$$\{a^n b^n \mid n \geq 0\}$$

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

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



Évitez 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.

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).

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 cette semaine 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.

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.

Trouver les non terminaux accessibles

- ▶ Règles :
 - ▶ l'axiome de la grammaires est accessible
 - ▶ Si N est accessible et $N \rightarrow u$ une production, alors tous les non terminaux en u sont accessibles.
- ▶ Exemple : $(\{a, b, c\}, \{A, B, C, D, E, F\}, F, P)$ avec P :

$A \rightarrow \epsilon \mid a$	$C \rightarrow A D \mid c$	$E \rightarrow A B$
$B \rightarrow \epsilon \mid b$	$D \rightarrow C a C$	$F \rightarrow E d E \mid A F$

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

Définition

- ▶ Soit $G = (\Sigma, N, S, P)$ une grammaire.
- ▶ 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 $K \rightarrow^* w$.
- ▶ Les non terminaux non accessibles ou non productifs sont inutiles.
- ▶ Une grammaire est *réduite* quand tous ses non-terminaux sont accessibles et productifs.

Trouver les non terminaux productifs

- ▶ Règles :
 - ▶ Si $N \rightarrow u$ est une production avec $u \in \Sigma^*$, alors N est productif.
 - ▶ Si $N \rightarrow u$ est une production et tous les non terminaux de u sont productifs, alors N est productif.
- ▶ Exemple : $(\{a, b, c\}, \{A, B, C, D, E, F\}, F, P)$ avec P :

$A \rightarrow \epsilon \mid a$	$C \rightarrow A D \mid c$	$E \rightarrow A B \mid B A$
$B \rightarrow E b$	$D \rightarrow C a C$	$F \rightarrow E d E \mid A F \mid c D$

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

Réduction d'une grammaire

- ▶ Pour toute grammaire G avec $\mathcal{L}(G) \neq \emptyset$ existe une grammaire G' telle que $\mathcal{L}(G) = \mathcal{L}(G')$ et G' est réduite.
- ▶ Construction :
 - ▶ on supprime d'abord tous les non-terminaux non productifs (et les règles dans lesquelles ils paraissent);
 - ▶ puis on supprime tous les non-terminaux non accessibles (et les règles dans lesquelles ils paraissent).
- ▶ Il faut le faire dans cet ordre.
- ▶ Quand $\mathcal{L}(G) = \emptyset$ alors l'axiome de la grammaire n'est pas productif, mais on n'a pas le droit de supprimer l'axiome !

Comment organiser une analyse descendante ?

- ▶ On construit l'arbre de dérivation à partir de sa racine, et puis on complète successivement par les enfants des nœuds étiquetés par des non-terminaux.
- ▶ Il faut maîtriser le choix de la règle de la grammaire qu'on applique .
- ▶ On ne peut pas demander qu'il y ait une seule règle par non-terminal (car dans ce cas la grammaire serait 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, où A est l'axiome :

$$A \rightarrow BC \mid D; D \rightarrow d; B \rightarrow b$$

- ▶ Non-terminal non productif : C . Le supprimer :
- ▶ On obtient

$$A \rightarrow D; D \rightarrow d; B \rightarrow b$$

- ▶ Non-terminal non accessible : B . Le supprimer :
- ▶ On obtient

$$A \rightarrow D; D \rightarrow d$$

- ▶ Cette grammaire est réduite.

Exemple

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

$$S \rightarrow i \quad (1)$$

$$S \rightarrow [S+S] \quad (2)$$

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

Construction d'un arbre de dérivation (1)

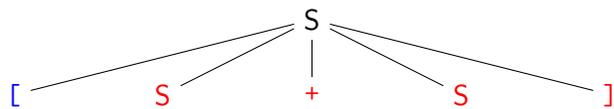
Productions : (1) $S \rightarrow i$ (2) $S \rightarrow [S+S]$
 S

[[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 (3)

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

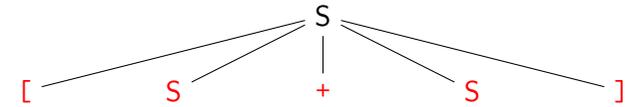


[[i + i] + [i + i]]

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) $S \rightarrow [S+S]$

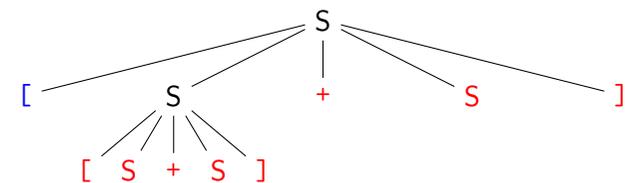


[[i + i] + [i + i]]

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

Construction d'un arbre de dérivation (4)

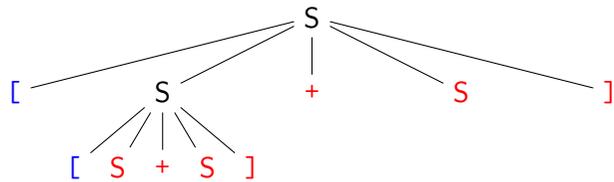
Productions : (1) $S \rightarrow i$ (2) $S \rightarrow [S+S]$



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

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

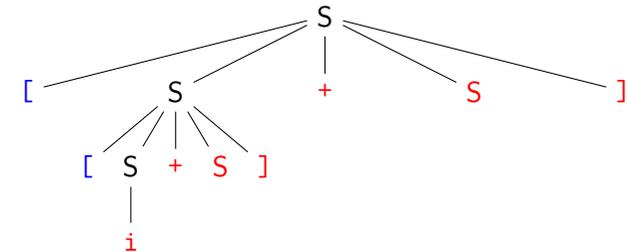
Construction d'un arbre de dérivation (5)

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

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

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

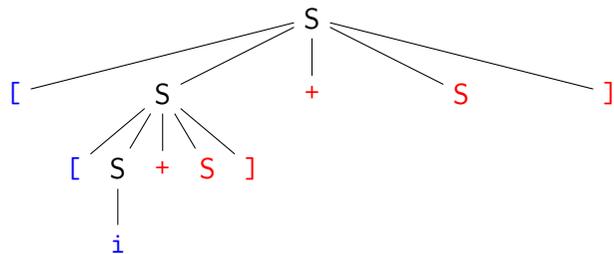
Construction d'un arbre de dérivation (6)

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

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

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

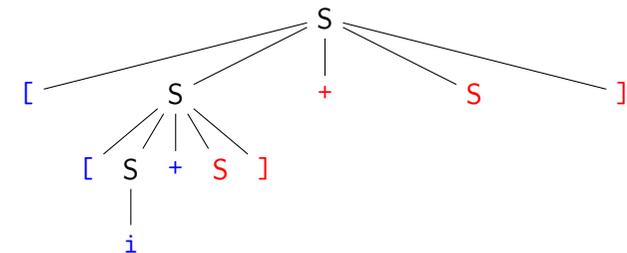
Construction d'un arbre de dérivation (7)

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

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

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

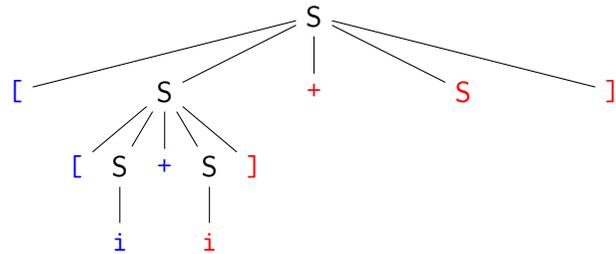
Construction d'un arbre de dérivation (8)

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

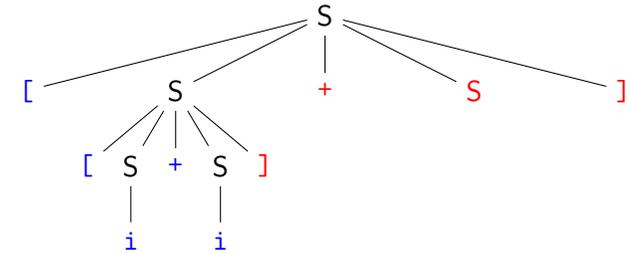
[[i	+	i]	+	[i	+	i]]
---	---	---	---	---	---	---	---	---	---	---	---	---

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

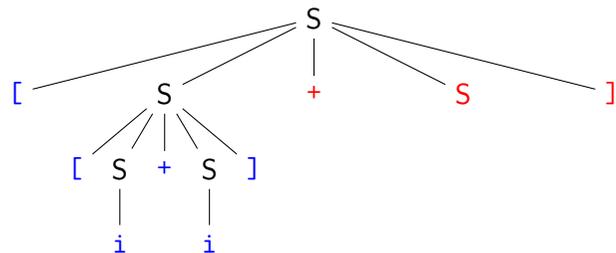
Construction d'un arbre de dérivation (9)

Productions : (1) $S \rightarrow i$ (2) $S \rightarrow [S+S]$ Le suivant non-terminal du mot des feuilles est i .

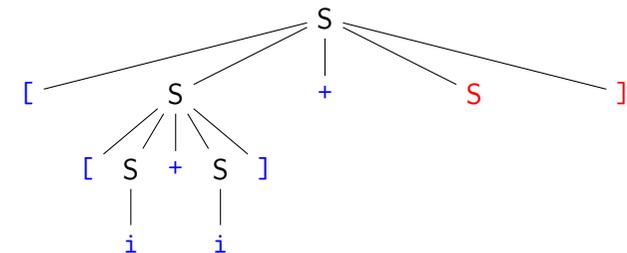
Construction d'un arbre de dérivation (10)

Productions : (1) $S \rightarrow i$ (2) $S \rightarrow [S+S]$ Le suivant non-terminal du mot des feuilles est $]$.

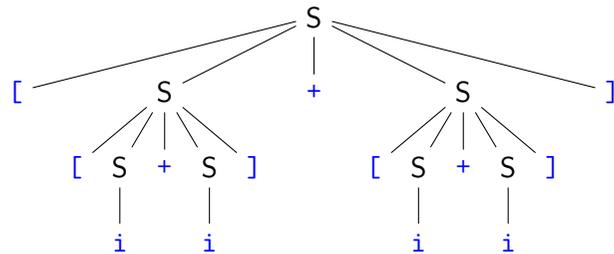
Construction d'un arbre de dérivation (11)

Productions : (1) $S \rightarrow i$ (2) $S \rightarrow [S+S]$ Seule la règle (2) peut produire à partir de S un mot qui commence par $+$.

Construction d'un arbre de dérivation (12)

Productions : (1) $S \rightarrow i$ (2) $S \rightarrow [S+S]$ Choisir règle (2) : c'est la seule qui peut produire à partir de S un mot qui commence par $[$.

etc. etc.



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 enfants 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 gauche (angl. : left) à 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)
- ▶ Généralisation des LL(1).
- ▶ En pratique ce sont surtout les grammaires LL(1) qui nous intéressent, pour cette raison le cas général LL(k) n'est pas fait dans ce cours.

Définition LL(1)

Définition

Soit $G = (\Sigma, N, S, P)$ une grammaire algébrique. G est dite **LL(1)** ssi

- ▶ S'il existe deux dérivations *gauches*

$$\begin{aligned} S &\rightarrow^* uY\alpha \rightarrow u\beta\alpha \rightarrow^* ux \\ S &\rightarrow^* uY\alpha \rightarrow u\gamma\alpha \rightarrow^* uy \end{aligned}$$

où $Y \in N$; $u, x, y \in \Sigma^*$; $\alpha \in (N \cup \Sigma)^*$; $Y \rightarrow \beta, Y \rightarrow \gamma \in P$,
 $\beta \neq \gamma$

- ▶ alors x et y diffèrent dans leur première lettre.

Conséquences

- ▶ Toute grammaire G qui est LL(1) 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(1) ?
- ▶ Notre définition parle de dérivations quelconques, ce n'est pas une méthode de décision efficace. Comment le décider efficacement en regardant la grammaire ?

Explication de la définition de LL(1)

- ▶ 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 le premier caractère de la suite du mot d'entrée, on peut maintenant décider comment réécrire le non-terminal Y .

Exemple d'une grammaire qui est LL(1)

- ▶ La grammaire de l'exemple précédent :

$$\begin{aligned} S &\rightarrow i \\ S &\rightarrow [S+S] \end{aligned}$$

est clairement LL(1) car les deux côtés droits de S commencent par des non-terminaux différents.

- ▶ La question est donc facile *quand tous les côtés droits des règles commencent par des terminaux* : pour tout non terminal, les côtés droits des règles pour *ce non-terminal* doivent commencer par des symboles différents.

La notion de First_1

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

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

- ▶ $\text{First}_1(\alpha)$ est l'ensemble des symboles par lesquels un mot terminal dérivé à partir de α peut commencer.
- ▶ Pour que G soit LL(1) il est nécessaire que quand il y a deux règles différentes $N \rightarrow \alpha$ et $N \rightarrow \beta$ pour le *même* non-terminal, alors $\text{First}_1(\alpha)$ est disjoint de $\text{First}_1(\beta)$.
- ▶ Si G n'a pas de règles de la forme $N \rightarrow \epsilon$ alors c'est aussi suffisant.

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

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

$$S \rightarrow i \quad (3)$$

$$S \rightarrow [S+S] \quad (4)$$

$$S \rightarrow [S*S] \quad (5)$$

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

Comment calculer le First_1 ?

- ▶ Quand le côté droit d'une règle commence par un terminal :

$$N \rightarrow c\alpha \quad (c \in \Sigma)$$

c'est trivial :

$$\text{First}_1(c\alpha) = \{c\}$$

- ▶ Exemple :

$$\text{First}_1(i) = \{i\}$$

$$\text{First}_1([S+S]) = \{[\}$$

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 suffisamment d'informations pour faire ce choix.
- ▶ Solution : Retardez le choix !
- ▶ par exemple, avec un non-terminal supplémentaire O :

$$S \rightarrow i \quad (6)$$

$$S \rightarrow [S O S] \quad (7)$$

$$O \rightarrow + \quad (8)$$

$$O \rightarrow * \quad (9)$$

Implémentation en OCaml

- ▶ L'analyseur grammatical (parser) consiste en plusieurs fonctions mutuellement récursives.
- ▶ Pour chaque non-terminal N on a une fonction (sans argument) qui essaye de lire une partie du texte d'entrée qui peut être dérivée de N .
Si c'est possible la fonction envoie le morceau d'arbre de dérivation correspondant, dans le cas contraire elle lève une exception.
- ▶ Dans chaque fonction il y a un cas par règle $N \rightarrow \alpha$. Chaque cas commence avec un test si le symbole suivant de l'entrée fait partie de $\text{First}_1(\alpha)$.

Fichier reader.mli |

```
(* 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

(* string representation of a token *)
val string_of_token: token -> string

(* 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
```

Implémentation en OCaml

- ▶ 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.
- ▶ Notre exemple :

$$\begin{aligned} S &\rightarrow [S + S] \\ S &\rightarrow i \end{aligned}$$

Axiome : S

Fichier tree.mli |

```
(* 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_S () =
  match lookahead () with
  | Ch 'i' -> begin (* S -> i *)
    eat (Ch 'i');
    Node("S" ,[Leaf 'i'])
  end
  | Ch '[' -> begin (* S -> [S+S] *)
    eat (Ch '[');
    let x1 = parse_S () in
    eat (Ch '+');
    let x2 = parse_S () in
    eat (Ch ']');
    Node("S" ,[Leaf '[';x1;Leaf '+'; x2; Leaf ']'])
```

Ce qui reste à faire

- ▶ Comment calculer le First_1 quand une règle commence par un non-terminal ?
- ▶ Est-il utile d'admettre des règles $N \rightarrow \epsilon$?
- ▶ Si oui, comment faire pour les traiter dans une analyse LL(1) ?

Fichier parser.ml II

```
end
| t -> raise (Error (Printf.sprintf
  "parsing S: unexpected %s"
  (Reader.string_of_token t)))

let parse () = parse_S ()
```