Langages de Programmation, Interprétation, Compilation

Christine Paulin
Christine.Paulin@universite-paris-saclay.fr

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

LDD3 Informatique, Mathématiques - Magistère Informatique

2025–26

Analyse syntaxique d'un langage impératif

Affichage d'une expression avec notations infixes

- Transformer un arbre de syntaxe abstraite en une chaîne représentant la même expression
- La structure d'arbre supprime le besoin de parenthèses
- L'affichage de l'arbre doit remettre des parenthèses
  - partout ? juste celles qui sont nécessaires ?
  - notion de priorité entre les opérateurs et d'associativité
  - plusieurs fonctions d'affichage de type expr –> string en fonction des priorités

Grammaires

La grammaire d'un langage décrit les structures de phrases légales dans ce langage.
On la définit à l'aide de deux éléments en plus du lexique :

- les différentes catégories de (fragments de) phrases du langage,
- les manières dont différents éléments peuvent être regroupés pour former un fragment de l'une ou l'autre des catégories (données par des règles de combinaisons).

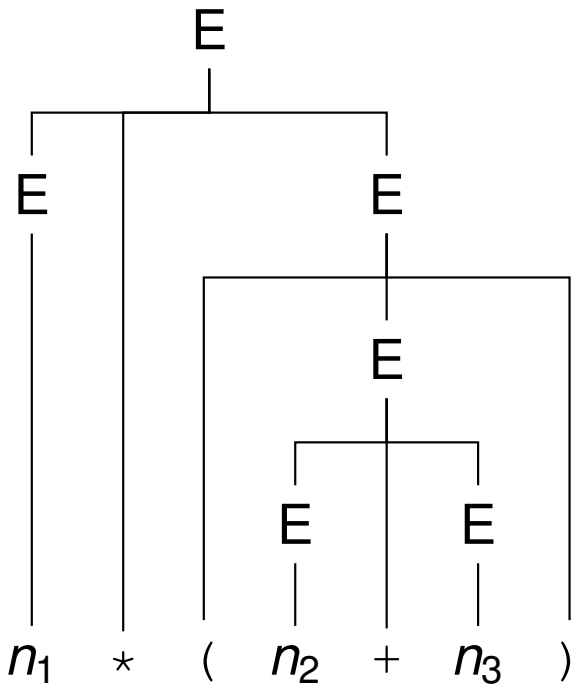
Les phrases bien formées sont alors celles dont les mots peuvent être regroupés d'une manière compatible avec les règles de combinaison.

Structure grammaticale d'une phrase

- les feuilles sont les mots de la phrase (dans l'ordre),
- chaque nœud interne dénote le regroupement de ses fils

$n_1 * (n_2 + n_3)$.

$$
\begin{array}{rcl}
E & ::= & n \\
 & | & E + E \\
 & | & E * E \\
 & | & (E)
\end{array}
$$



Grammaire algébrique

Une grammaire algébrique est un quadruplet (T, N, S, R) où :

- T est un ensemble de symboles terminaux, désignant des mots,
- N est un ensemble de symboles non terminaux, désignant des groupes de mots,
- S est le symbole de départ, qui décrit une phrase complète,
- R est un ensemble de règles de production
  - paires $X \prec a_1 \dots a_k$ avec X symbole non terminal et $a_1 \dots a_k$ de symboles (terminaux ou non) formant un groupe de type X correct (qui peut être vide).

Grammaire simple pour les expressions arithmétiques

- les symboles terminaux `+`, `*`, `(`, `)` ainsi qu'un terminal `n` pour une constante entière,
- le symbole non terminal E,
- le symbole de départ E,
- les règles de production $E \prec \texttt{n}$, $E \prec E\texttt{+}E$, $E \prec E\texttt{*}E$ et $E \prec \texttt{(}E\texttt{)}$, encore écrites

$$\begin{array}{rcl} E & ::= & \texttt{n} \\ & | & E\ \texttt{+}\ E \\ & | & E\ \texttt{*}\ E \\ & | & \texttt{(}\ E\ \texttt{)} \end{array}$$

Arbre de dérivation syntaxique

- nœud interne : symbole de N,
- feuille : symbole de T,
- si un nœud interne a l'étiquette $X \in N$, les étiquettes de ses fils prises dans l'ordre forment une séquence β telle que $X \prec \beta \in R$

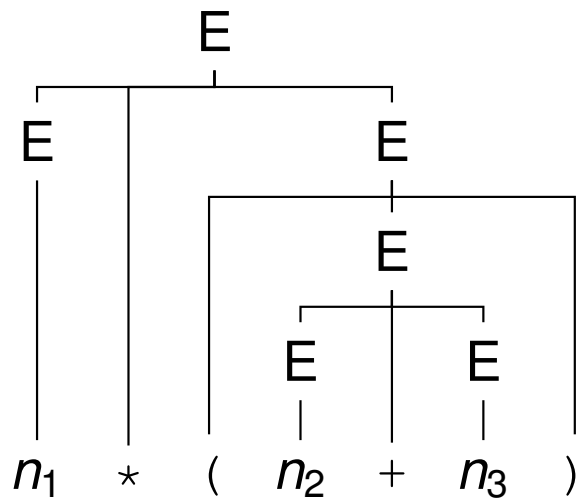
Type Caml des arbres de dérivation (paramétré par le type des terminaux) :

```
type 't deriv =
        Node of string * 't deriv list
      | Leaf of 't
```

Une phrase m est dérivable à partir d'un symbole non terminal X s'il existe un arbre de dérivation dont la racine est étiquetée par X et dont les feuilles, prises dans l'ordre, forment cette phrase.
Les phrases dérivables de la grammaire sont celles qui sont dérivables à partir du symbole de départ S.

Exemple

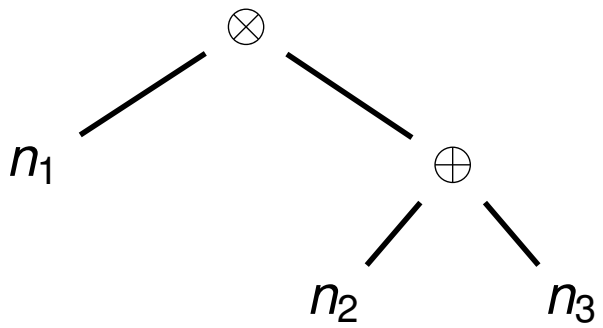
Arbre de dérivation pour l'expression arithmétique $n_1 * (n_2 + n_3)$ avec la grammaire naïve



Arbre de syntaxe abstraite

Ne pas confondre arbre de dérivation avec l'arbre de syntaxe abstraite

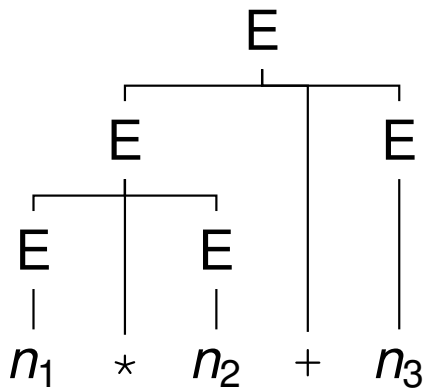
```
Mul(Cst n1, Add(Cst n2, Cst n3))
```

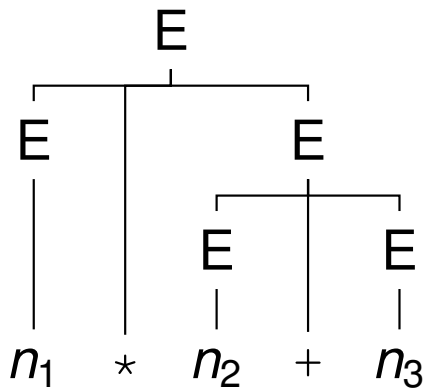


- L'analyseur syntaxique construit l'arbre de syntaxe abstraite.
- L'arbre de dérivation décrit les différentes étapes de l'analyse réalisée, mais n'est pas construit explicitement par l'analyseur.

Ambiguïtés

Une grammaire est ambiguë s'il existe une phrase qui peut être dérivée par deux arbres différents.
Pour la phrase $n_1 \star n_2 + n_3$:



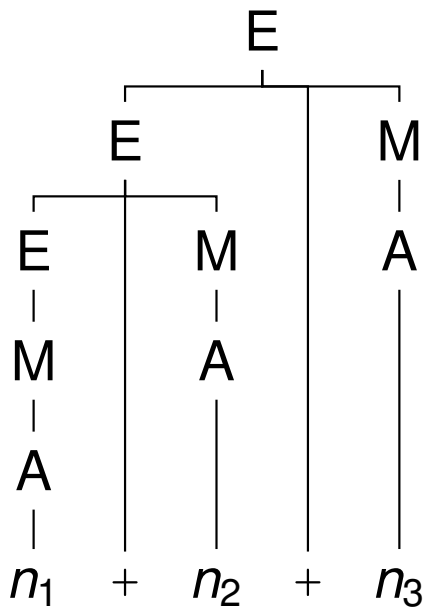


Grammaire non-ambigüe

Reconnait le même langage

$$
\begin{array}{lcl}
E & ::= & E + M \\
 & | & M \\
M & ::= & M * A \\
 & | & A \\
A & ::= & n \\
 & | & (E)
\end{array}
$$

Arbre de dérivation pour la phrase $n_1 + n_2 + n_3$.



Partie 1 : Introduction

Analyseur syntaxique

L'analyseur syntaxique

- prend en entrée une séquence de lexèmes,
- doit vérifier que cette séquence est cohérente avec les règles de grammaire du langage et renvoie
  - en cas de succès, un arbre de syntaxe abstraite donnant la structure de la séquence lue en entrée,
  - en cas d'échec, la localisation et l'explication de ce qui est grammaticalement incohérent dans l'entrée.

Analyse descendante

- Construire l'arbre de la racine vers les feuilles pour arriver à la suite de terminaux passée en entrée. On parle d'analyse descendante, ou top-down
- On définit pour chaque symbole non terminal X, une fonction f_X :
  - prend en entrée une séquence de lexèmes l,
  - cherche dans un préfixe de l une phrase m pour une règle $X \prec m$,
  - renvoie les lexèmes qui n'ont pas été utilisés
  - échoue si elle ne parvient pas à reconnaître de fragment m pour aucune règle

Construction manuelle d'un analyseur

Grammaire non ambiguë des expressions arithmétiques

$$
\begin{array}{lcl}
E & ::= & E + M \\
 & | & M \\
M & ::= & M * A \\
 & | & A \\
A & ::= & n \\
 & | & (E)
\end{array}
$$

Type de données pour les lexèmes

```
type token =
  | NUM of int | PLUS | STAR | LPAR | RPAR
```

Reconnaissance d'un atome

Reconnaître un atome en préfixe d'une suite de lexèmes :

```
let validate_atom: token list -> token list = function
```

- Le premier lexème est soit une constante soit une parenthèse
- Si constante `n`, on réussit et on renvoie le reste de la liste
- Si parenthèse ouvrante `(`, alors la règle qui peut s'appliquer est $A \prec (E)$. Reconnaître une expression puis vérifier la parenthèse fermante

```
let rec validate_atom = function
    | NUM _ :: l -> l
    | LPAR :: l -> let l' = validate_expr l in
                   let l'' = match l' with
                     | RPAR :: l'' -> l''
                     | _ -> failwith "syntax error"
                   in
                   l''
    | _ -> failwith "syntax error"
```

Changement de grammaire

Pouvoir faire le choix de la règle en regardant le premier léxème de l'entrée

$$
\begin{array}{lcl}
E & ::= & M\ E' \\
E' & ::= & +\ E \\
 & | & \varepsilon \\
M & ::= & A\ M' \\
M' & ::= & *\ M \\
 & | & \varepsilon \\
A & ::= & n \\
 & | & (\ E\ )
\end{array}
$$

Partie 1 : Introduction

Analyseur complet pour IMP : symboles

Symboles terminaux :

- un symbole *n* pour chaque constante entière,
- un symbole *x* pour chaque nom de variable,
- les symboles `+`, `*`, `<`, `&&`, `(`, `)`, `{`, `}`, `:=`, `;`,
- les mots-clés `print`, `while`, `if`, `else`.

Symboles non-terminaux

- Symbole de départ *P* : une séquence d'instructions.
- Symbole *S* pour une suite d'instructions (peut être vide)
- Symbole *I* pour une instruction
- Symbole *E* pour une expression (opération binaire ou expression atomique)
- Symbole *A* pour une expression atomique (constante, variable ou expression parenthésée)

Analyseur complet pour IMP : Règles

$$
\begin{array}{lcll}
P & ::= & S & \\
S & ::= & \varepsilon & \\
 & | & I\ S & \\
I & ::= & x\ \texttt{:=}\ E\ \texttt{;} & \\
 & | & \texttt{while (}\ E\ \texttt{) \{}\ S\ \texttt{\}} & \\
 & | & \texttt{if (}\ E\ \texttt{) \{}\ S\ \texttt{\} else \{}\ S\ \texttt{\}} & \\
 & | & \texttt{print (}\ E\ \texttt{);} & \\
E & | & A & \\
 & | & E\ o\ E & \text{avec } o \in \{+, *, <, \&\&\} \\
A & ::= & n & \\
 & | & x & \\
 & | & \texttt{(}\ E\ \texttt{)} &
\end{array}
$$

Analyse lexicale

```
type token =
  | NUM of int                          (* n                    *)
  | VAR of string                       (* x                    *)
  | PLUS | STAR | LT | AND              (* + * < &&             *)
  | LPAR | RPAR | BEGIN | END           (* ( ) { }              *)
  | SET | SEMI                          (* := ;                 *)
  | PRINT | WHILE | IF | ELSE
  | EOI

let tokenizer (s: string): unit -> token =
  let i = ref 0 in
  let rec next_token () = match s.[!i] with
    | '+' -> incr i; PLUS
    | '*' -> incr i; STAR
.........
    | ':' -> if s.[!i+1] = '=' then (i := !i+2; SET)
             else failwith "lexical error"
    | '&' -> if s.[!i+1] = '&' then (i := !i+2; AND)
             else failwith "lexical error"
```

Analyse lexicale mots clés

```
  | 'a'..'z' -> let k = scan_word 1 in
                let t = String.sub s !i k in
                i := !i + k;
                 (match t with
                  | "print" -> PRINT
                  | "while" -> WHILE
                  | "if"    -> IF
                  | "else"  -> ELSE
                  | _       -> VAR t)
  | ' ' | '\n' -> incr i; next_token ()
  | c -> failwith (Printf.sprintf "unknown character %c" c)
  | exception e -> EOI
and scan_word k = match s.[!i+k] with
  | 'a'..'z' -> scan_word (k+1)
  | _ -> k
  | exception e -> k
```

Syntaxe abstraite

```
type prog = instr list
and instr =
    | Print of expr
    | Set   of string * expr
    | While of expr * instr list
    | If    of expr * instr list * instr list
and expr =
    | Cst of int
    | Var of string
    | Bop of bop * expr * expr
and bop = Add | Mul | Lt | And
```

On associe à chaque opérateur une priorité

```
let prio = function
    | AND  -> 1 | LT -> 2 | PLUS -> 3 | STAR -> 4
    | _    -> failwith "prio: expected operator"
let root_prio = 0
```

Analyse syntaxique

- Une fonction interne tokenizer qui lit le prochain léxème
- une référence current_token pour le léxème courant
- next : unit –> unit : mise à jour par par le prochain léxème
- take_token : unit –> token : renvoie le léxème courant et lit le suivant
- expect (s : token) : unit : vérifie que le léxème courant est s et lit le suivant

Analyse descendante : suite d'instructions

Suite d'instructions, peut se terminer par la fin de l'entrées `EOI` ou par l'accolade fermante

```
let rec parse_instrs () = match !current_token with
    | EOI | END -> []
    | _ -> let i = parse_instr () in
           let is = parse_instrs () in
           i :: is
```

Analyse descendante : instruction

Pour une instruction, le premier lexème détermine la règle à appliquer

```
and parse_instr () = match take_token() with
    | VAR x -> expect SET;
               let e = parse_expr root_prio in
               expect SEMI;
               Set(x, e)
    | PRINT -> let e = parse_cond () in
                  expect SEMI;
                  Print(e)
    | WHILE -> let e = parse_cond () in
                  let is = parse_block () in
                  While(e, is)
    | IF    -> let e = parse_cond () in
                  let is1 = parse_block () in
                  expect ELSE;
                  let is2 = parse_block () in
                  If(e, is1, is2)
                  | _ -> failwith "syntax␣error"
```

Analyse descendante : instruction (suite)

```
and parse_cond () =
      expect LPAR;
      let e = parse_expr root_prio in
      expect RPAR;
      e
and parse_block () =
      expect BEGIN;
      let is = parse_instrs () in
      expect END;
      is
```

Analyse descendante : expressions

Utilisation de fonctions paramétrées par le niveau de priorité

- parse_expr p : reconnaît une expression dont l'opérateur principal est d'une priorité $> p$
- expand_op p : cherche à compléter l'expression par des opérateurs de priorité $> p$
- initialisé par root_prio (plus petit que tous les opérateurs)

(on parle d'analyse par priorit és ascendantes). À l'initialisation, on utilise une priorité root_prio qui est le niveau le plus bas.

```
and parse_a_expr () = match take_token() with
      | LPAR -> let e = parse_expr root_prio in
                expect RPAR;
                e
      | NUM n -> Cst n
      | VAR x -> Var x
      | _ -> failwith "syntax␣error"
```

Analyse descendante : expressions (code)

```
and parse_expr p = parse_a_expr () |> expand_op p

and expand_op p e1 = match !current_token with
  | (PLUS | STAR | LT | AND as op) when (prio op > p) –
      next();
      let e2 = parse_expr (prio op) in
      let e = match op with
        | PLUS –> Bop(Add, e1, e2)
        | STAR –> Bop(Mul, e1, e2)
        | LT   –> Bop(Lt, e1, e2)
        | AND  –> Bop(And, e1, e2)
        | _    –> assert false
      in
      expand_op p e
  | _ –> e1
```

Partie 1 : Introduction

Dérivations

- grammaire (T, N, S, R)
- suite de phrases, du symbole de départ à une suite de terminaux
- Définition : $m \to m'$ si
  - $m = m_1 X m_2$, $m' = m_1 a_1 \dots a_n m_2$
  - $X \prec a_1 \dots a_k \in R$
- Fermeture transitive : $m \to^+ m'$ (au moins une étape)
- Fermeture réflexive-transitive : $m \to^* m'$ (0 ou plusieurs étapes)
- Une séquence m est dérivable lorsqu'il existe une dérivation $S \to^* m$.
- On parle de dérivation gauche lorsque on expanse à chaque étape le symbole non terminal le plus à gauche.
- Il n'y a plus de dérivation possible à partir d'une suite de terminaux
- Plusieurs dérivation peuvent donner le même arbre (mais une seule dérivation gauche)

Exemples

$E * (E)$ se dérive en $E * (E+E)$ en une étape

$$\begin{aligned} m_1 &= E * (\\ m_2 &=) \\ X &= E \\ \beta &= E+E \end{aligned}$$

Dérivation de $\mathtt{n}_1 \ * \ (\mathtt{n}_2 \ + \ \mathtt{n}_3)$:

$$\begin{aligned} E &\to E * E \\ &\to \mathtt{n}_1 * E \\ &\to \mathtt{n}_1 * (E) \\ &\to \mathtt{n}_1 * (E \ + \ E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 \ + \ E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 \ + \ \mathtt{n}_3) \end{aligned}$$

Partie 1 : Introduction

Construction systématique d'un analyseur descendant

- Systématiser la construction de l'analyseur à partir de la grammaire
- Pour un non terminal X choisir la règle $X \prec \beta$ à expanser de manière déterministe en fonction de l'entrée
  - possible seulement si la grammaire est non ambigüe
- Faire ce choix en fonction uniquement du premier caractère de l'entrée !

(Couvert dans le cours de PIL de L2)

Premiers et annulables

- grammaire (T, N, S, R),
- séquence $\alpha \in (T \cup N)^*$ de symboles terminaux ou non terminaux.
- On note Premiers(α) l'ensemble des symboles terminaux qui peuvent se trouver en tête d'une séquence dérivée à partir de α.

$$\text{Premiers}(\alpha) = \{\, a \in T \mid \exists \beta, \alpha \to^* a\beta \,\}$$

- On note Annulables l'ensemble des symboles non terminaux à partir desquels il est possible de dériver la séquence vide :

$$\text{Annulables} = \{\, X \in N \mid X \to^* \varepsilon \,\}$$

Caractérisation récursive

Premiers

$$
\begin{array}{rcll}
\text{Premiers}(\varepsilon) & = & \emptyset & \\
\text{Premiers}(a\beta) & = & \{a\} & \text{si } a \in T \\
\text{Premiers}(X) & = & \bigcup_{X \prec \beta \in R} \text{Premiers}(\beta) & \\
\text{Premiers}(X\beta) & = & \begin{cases} \text{Premiers}(X) & \text{si } X \notin \text{Annulables} \\ \text{Premiers}(X) \cup \text{Premiers}(\beta) & \text{si } X \in \text{Annulables} \end{cases} &
\end{array}
$$

Annulables Le symbole X est annulable :

- s'il existe une règle $X \prec \varepsilon$, ou
- s'il existe une règle $X \prec X_1 \ldots X_k$ où tous les X_i sont des symboles non terminaux qui sont également annulables.

Equations récursives dont on cherche le plus petit point fixe.

Théorème de point fixe de Tarski (fini)

Si on considère un ensemble ordonné A fini et doté d'un plus petit élément $\bot$, et une fonction $F : A \to A$ croissante, alors :

- la fonction F admet au moins un point fixe, c'est-à-dire qu'il existe un élément $x \in A$ tel que $F(x) = x$, et
- en itérant F à partir de $\bot$, c'est-à-dire en calculant $F(\bot)$, puis $F(F(\bot))$, puis $F(F(F(\bot)))$, etc, on finit nécessairement par trouver un point fixe de F (et même plus précisément : le plus petit des points fixes).

Ce théorème est vrai dans un cadre plus large (existence d'une borne sup pour toute chaîne croissante)

Calcul des annulables

- On cherche un ensemble de symboles non terminaux.
- A est l'ensemble des parties de N, ordonné par l'ordre d'inclusion.
- Le plus petit élément est $\emptyset$.
- La fonction croissante est $F : \mathcal{P}(N) \to \mathcal{P}(N)$ définie par

$$F(E) = \{ X \mid X \to \varepsilon \} \cup \{ X \mid X \to X_1 \dots X_k,\ X_i \in E \}$$

- Exemple des expressions arithmétiques : $F(\emptyset) = \{ E', M' \}$ et $F(\{ E', M' \}) = \{ E', M' \}$.
  Le point fixe est $\{ E', M' \}$, seuls symboles non terminaux annulables.
- Présenté dans un tableau de booléens donnant les annulables trouvés à chaque itération.
  On s'arrête lorsque deux lignes consécutives sont identiques.

	E	E'	M	M'	A
0.	F	F	F	F	F
1.	F	V	F	V	F
2.	F	V	F	V	F

Calcul des premiers

- Calculer pour chaque symbole non terminal, un ensemble de symboles terminaux.
- On considère l'ensemble $\mathcal{P}(T) \times \ldots \times \mathcal{P}(T)$ (avec une composante par symbole non terminal), ordonné par le produit de l'ordre inclusion.
- Son plus petit élément est $(\emptyset, \ldots, \emptyset)$,
- la fonction croissante utilisée est celle qui recalcule les informations pour chaque symbole non terminal en fonction des informations déjà connues.
- Présentation dans un tableau, chaque colonne correspond à un symbole non terminal, et chaque ligne à une itération.
- Le calcul s'arrête lorsque deux lignes deviennent identiques.

	E	E'	M	M'	A
0.	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1.	$\emptyset$	$\{ + \}$	$\emptyset$	$\{ * \}$	$\{ n, (\}$
2.	$\emptyset$	$\{ + \}$	$\{ n, (\}$	$\{ * \}$	$\{ n, (\}$
3.	$\{ n, (\}$	$\{ + \}$	$\{ n, (\}$	$\{ * \}$	$\{ n, (\}$
4.	$\{ n, (\}$	$\{ + \}$	$\{ n, (\}$	$\{ * \}$	$\{ n, (\}$

Suivants

- une règle $X \prec \beta$ peut s'appliquer si le premier terminal de l'entrée appartient à Premiers(β)
- que se passe-t-il si $\beta \to^* \varepsilon$ (en particulier si $\beta = \varepsilon$) ?
- le premier symbole de l'entrée est alors un symbole qui peut suivre X dans une dérivation
- En analysant la grammaire, on peut caractériser l'ensemble des suivants d'un symbole non terminal $X \in N$: peut apparaître immédiatement après X dans une dérivation.

$$\text{Suivants}(X) = \{\, a \in T \mid \exists \alpha, \beta,\ S \to^* \alpha X a \beta \,\}$$

- Exemple des expressions arithmétiques :
  - $S \to M\ E' \to M + E$ et

$$S \to M\ E' \to M + E \to A\ M' + E \to A\ \varepsilon + E = A + E$$

$+ \in \text{Suivants}(M)$ et $+ \in \text{Suivants}(A)$

Equations pour les suivants

Pour calculer l'ensemble des suivants de X, on regarde toutes les apparitions de X dans un membre droit de règle $Y \prec \alpha X \beta$

- on inclut dans Suivants(X) :
  - les symboles terminaux qui peuvent apparaître en tête d'une séquence dérivée à partir de β, c'est-à-dire Premiers(β),
  - dans le cas où $\beta \rightarrow^* \varepsilon$ tous les suivants de Y peuvent également suivre X

On a à nouveau des équations récursives, à résoudre par la même technique que précédemment.

Calcul des suivants

On ajoute le symbole terminal spécial $\#$ de fin d'entrée (correspond à EOI), il appartient aux suivants du symbole de départ S.
Dans notre exemple des expressions arithmétiques :

$$\begin{aligned}
\text{Suivants}(E) &= \{\#,)\} \cup \text{Suivants}(E') \\
\text{Suivants}(E') &= \text{Suivants}(E) \\
\text{Suivants}(M) &= \{+\} \cup \text{Suivants}(E) \cup \text{Suivants}(M') \\
\text{Suivants}(M') &= \text{Suivants}(M) \\
\text{Suivants}(A) &= \{\star\} \cup \text{Suivants}(M)
\end{aligned}$$

	E	E'	M	M'	A
0.	$\{\#\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1.	$\{\#,)\}$	$\{\#\}$	$\{\#, +\}$	$\emptyset$	$\{\star\}$
2.	$\{\#,)\}$	$\{\#,)\}$	$\{\#, +,)\}$	$\{\#, +\}$	$\{\#, +, \star\}$
3.	$\{\#,)\}$	$\{\#,)\}$	$\{\#, +,)\}$	$\{\#, +,)\}$	$\{\#, +, \star,)\}$
4.	$\{\#,)\}$	$\{\#,)\}$	$\{\#, +,)\}$	$\{\#, +,)\}$	$\{\#, +, \star,)\}$

Analyse LL

Avec ces éléments, on peut construire une table d'analyse LL(1).
Pour chaque non terminal X (en cours d'analyse) et symbole terminal a (en tête de l'entrée), on cherche la(les) règle(s) à appliquer
Le principe de construction est le suivant :

- Pour chaque règle $X \prec \beta$ et chaque symbole terminal $a \in$ Premiers(β) on indique la règle $X \prec \beta$ dans la case de la ligne X et de la colonne a.
- Pour chaque règle $X \prec \beta$ avec $\beta \rightarrow^* \varepsilon$ et chaque symbole terminal $a \in$ Suivants(X) on indique la règle $X \prec \beta$ dans la case de la ligne X et de la colonne a.

Si chaque case contient au plus une règle, on obtient une table d'analyse déterministe.
La grammaire utilisée est dite LL(1)

Exemple

	n	$+$	$\star$	$($	$)$	$\#$
E	$M\ E'$			$M\ E'$		
E'		$+\ E$			ε	ε
M	$A\ M'$			$A\ M'$		
M'		ε	$\star\ M$		ε	ε
A	n			$(\ E\ )$		

Conflits

- Si une case de la table contient plusieurs règles, on dit que la table contient un conflit, et que la grammaire n'est pas LL(1).
- Pour obtenir un analyseur dans ce cas, il faut revoir la grammaire ou utiliser une autre technique.
- être LL(1) est une propriété de la grammaire, et non du langage décrit par la grammaire.
- une grammaire LL(1) est non-ambigüe

Bilan sur l'analyse récursive descendante

- La technique d'analyse descendante est facile à programmer
- Contraintes fortes sur la grammaire pour choisir la règle à appliquer
  - fonctionne bien pour les instructions qui débutent par un mot clé
  - moins naturel pour les notations infixes (comme les opérateurs dans les expressions)
- D'autres techniques existent, avec des caractéristiques différentes.

Synthèse

A retenir

- Définition d'une grammaire, d'un arbre de dérivation syntaxique
- Notion d'ambiguïté
- Principales grammaires pour les expressions arithmétiques (grammaires naturelles ambigüe, non-ambigüe, grammaire LL(1)), grammaires pour des séquences d'instructions (séparées ou non par un caractère)
- Différence entre arbre de dérivation syntaxique et arbre de syntaxe abstraite
- Schéma général de l'analyse descendante (recherche du prochain lexème, fonctions mutuellement récursives de reconnaissance de non-terminaux)
- Analyseur descendant pour des expressions arithmétiques avec priorité

Savoir faire

- Adapter l'analyseur descendant de IMP à de nouvelles constructions