# Introduction à la compilation – TD 1 : Analyse LL(1)

Université Paris Diderot – Licence 3

(2012-2013)

**Exercice 1**

$$S \to E\$$$
$$E \to OEE \mid A$$
$$O \to + \mid *$$
$$A \to n \mid (E)$$

1. *Calculez la fonction FIRST pour tous les non terminaux de cette grammaire.*

   Réponse:
   - FIRST (S) = FIRST (E) = { +, *, (, n }
   - FIRST (O) = { +, * }
   - FIRST (A) = { n, ( }

2. *La grammaire est-elle LL(1) ? Si oui, quelle est la table d'analyse correspondante ? Si non, existe-t-il une grammaire équivalente LL(1) ? Dans ce cas, donnez-en la table d'analyse.*

   Réponse: Pour savoir si une grammaire est LL(1), il suffit d'essayer de construire la table d'analyse LL(1) de la grammaire. Ainsi, on obtient :

| | $*$ | $+$ | $($ | $)$ | $n$ | $\$$ |
|---|---|---|---|---|---|---|
| $S$ | $S \to E\$$ | $S \to E\$$ | $S \to E\$$ | $\emptyset$ | $S \to E\$$ | $\emptyset$ |
| $E$ | $E \to OEE$ | $E \to OEE$ | $E \to A$ | $\emptyset$ | $E \to A$ | $\emptyset$ |
| $O$ | $O \to *$ | $O \to +$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $A$ | $\emptyset$ | $\emptyset$ | $A \to (E)$ | $\emptyset$ | $A \to n$ | $\emptyset$ |

   Il n'y a qu'une règle par case, cette grammaire est bien LL(1).

3. *Écrivez un programme CAML qui interprète cette table pour déterminer si un mot est dans le langage engendré par cette grammaire.*

   Réponse:

```
type nonterminal = S | E | O | A
type terminal = LPAREN | RPAREN | EOF | INT | PLUS | STAR

type symbol = Terminal of terminal | NonTerminal of nonterminal

type rule =
  | Rule1 (* S -> E $ *)
  | Rule2 (* E -> O E E *)
  | Rule3 (* E -> A *)
  | Rule4 (* O -> + *)
  | Rule5 (* O -> * *)
  | Rule6 (* A -> n *)
```

```
  | Rule7 (* A -> ( E ) *)

let expand = function
  | Rule1 → [NonTerminal E ; Terminal EOF ]
  | Rule2 → [NonTerminal O ; NonTerminal E ; NonTerminal E]
  | Rule3 → [NonTerminal A]
  | Rule4 → [Terminal PLUS]
  | Rule5 → [Terminal STAR]
  | Rule6 → [Terminal INT]
  | Rule7 → [Terminal LPAREN ; NonTerminal E ; Terminal RPAREN]

exception Fail

let table = function
  | S → (function
      | STAR | PLUS | LPAREN | INT → Rule1
      | _ → raise Fail)
  | E → (function
      | STAR | PLUS → Rule2
      | LPAREN | INT _ → Rule3
      | _ → raise Fail)
  | O → (function
      | STAR → Rule5
      | PLUS → Rule4
      | _ → raise Fail)
  | A → (function
      | LPAREN → Rule7
      | INT _ → Rule6
      | _ → raise Fail)

type prediction = symbol list

type input = terminal list

type configuration = input * prediction

exception TokenError of terminal * terminal
exception SyntaxError

let rec parse = function
  | (token :: input, Terminal token' :: prediction) →
    if token = token' then
      parse (input, prediction)
    else
      raise (TokenError (token, token'))

  | (token :: input, NonTerminal s :: prediction) →
    let rule = table s token in
    parse (token :: input, expand rule @ prediction)

  | ([], []) →
    ()
```

```
    | _ →
      raise SyntaxError

let parse input = parse (input, [NonTerminal S])
```

4. *Écrivez un programme CAML qui implémente cette table à l'aide de quatre fonctions mutuellement récursives.*
   Réponse:

```
let parse =
  let ( |> ) p1 p2 = fun input → p2 (p1 input)
  and ( » ) input p = p input
  in
  let current_token = function
    | token :: _ → token
    | [] → assert false
  in
  let accept token input =
    if current_token input ≠ token then
      raise (TokenError (current_token input, token));
    List.tl input
  in

  let rec parse_S input =
    match current_token input with
      | STAR | PLUS | LPAREN | INT → input » (parse_E |> (accept EOF))
      | _ → raise SyntaxError

  and parse_E input =
    match current_token input with
      | STAR | PLUS → input » (parse_O |> (parse_E |> parse_E))
      | LPAREN | INT → input » parse_A
      | _ → raise SyntaxError

  and parse_O input =
    match current_token input with
      | STAR → input » (accept STAR)
      | PLUS → input » (accept PLUS)
      | _ → raise SyntaxError

  and parse_A input =
    match current_token input with
      | LPAREN → input » (accept LPAREN |> (parse_E |> (accept RPAREN)))
      | INT → input » (accept INT)
      | _ → raise SyntaxError
  in
  parse_S
```

5. *Explicitez les étapes de la reconnaissance de la chaîne "*$+(*21)(+(+11)2)\$$*".*
   Réponse:

| | | | |
|---|---|---|---|
| $+(*II)(+(+II)I)\$$ | $II)(+(+II)I)\$$ | $+(+II)I)\$$ | $II)I)\$$ |
| $S$ | $EE)E\$$ | $E)\$$ | $AE)E)\$$ |
| $+(*II)(+(+II)I)\$$ | $II)(+(+II)I)\$$ | $+(+II)I)\$$ | $II)I)\$$ |
| $E\$$ | $AE)E\$$ | $OEE)\$$ | $IE)E)\$$ |
| $+(*II)(+(+II)I)\$$ | $II)(+(+II)I)\$$ | $+(+II)I)\$$ | $I)I)\$$ |
| $OEE\$$ | $IE)E\$$ | $+EE)\$$ | $E)E)\$$ |
| $+(*II)(+(+II)I)\$$ | $I)(+(+II)I)\$$ | $(+II)I)\$$ | $I)I)\$$ |
| $+EE\$$ | $E)E\$$ | $EE)\$$ | $A)E)\$$ |
| $(*II)(+(+II)I)\$$ | $I)(+(+II)I)\$$ | $(+II)I)\$$ | $I)I)\$$ |
| $EE\$$ | $A)E\$$ | $AE)\$$ | $I)E)\$$ |
| $(*II)(+(+II)I)\$$ | $I)(+(+II)I)\$$ | $(+II)I)\$$ | $)I)\$$ |
| $AE\$$ | $I)E\$$ | $(E)E)\$$ | $)E)\$$ |
| $(*II)(+(+II)I)\$$ | $)(+(+II)I)\$$ | $+II)I)\$$ | $I)\$$ |
| $(E)E\$$ | $)E\$$ | $E)E)\$$ | $E)\$$ |
| $*II)(+(+II)I)\$$ | $(+(+II)I)\$$ | $+II)I)\$$ | $I)\$$ |
| $E)E\$$ | $E\$$ | $OEE)E)\$$ | $A)\$$ |
| $*II)(+(+II)I)\$$ | $(+(+II)I)\$$ | $+II)I)\$$ | $I)\$$ |
| $OEE)E\$$ | $A\$$ | $+EE)E)\$$ | $I)\$$ |
| $*II)(+(+II)I)\$$ | $(+(+II)I)\$$ | $II)I)\$$ | $)\$$ |
| $*EE)E\$$ | $(E)\$$ | $EE)E)\$$ | $)\$$ |

| |
|---|
| $\$$ |
| $\$$ |

6. *Explicitez les étapes du rejet de la chaîne "$(+1(*2))\$$".*
   Réponse:

| | | | |
|---|---|---|---|
| $(+I(*I))\$$ | $+I(*I))\$$ | $(*I))\$$ | $*I))\$$ |
| $S$ | $OEE)\$$ | $E)\$$ | $*EE))\$$ |
| $(+I(*I))\$$ | $+I(*I))\$$ | $(*I))\$$ | $I))\$$ |
| $E\$$ | $+EE)\$$ | $A)\$$ | $EE))\$$ |
| $(+I(*I))\$$ | $I(*I))\$$ | $(*I))\$$ | $I))\$$ |
| $A\$$ | $EE)\$$ | $(E))\$$ | $AE))\$$ |
| $(+I(*I))\$$ | $I(*I))\$$ | $*I))\$$ | $I))\$$ |
| $(E)\$$ | $AE)\$$ | $E))\$$ | $IE))\$$ |
| $+I(*I))\$$ | $I(*I))\$$ | $*I))\$$ | $))\$$ |
| $E)\$$ | $IE)\$$ | $OEE))\$$ | $E))\$$ |

7. *( ) Comment adapter votre analyseur syntaxique pour qu'il produise un arbre de production si il existe ?*
   Réponse:

```
type action =
  | Recognize of symbol
```

```
  | Produce of rule

let recognize_using r = List.map (fun x → Recognize x) (expand r) @ [Produce r]

type prediction = action list

type input = terminal list

type ptree =
  | Rule of rule * ptree list
  | Leaf of terminal

exception TokenError of terminal * terminal
exception SyntaxError

let apply_rule = function
  | Rule1 → (function
      | tEOF :: tE :: ts → Rule (Rule1, [tE; tEOF]) :: ts
      | _ → raise SyntaxError)
  | Rule2 → (function
      | tE :: tE' :: tO :: ts → Rule (Rule2, [tO; tE'; tE]) :: ts
      | _ → raise SyntaxError)
  | Rule3 → (function
      | tA :: ts → Rule (Rule3, [tA]) :: ts
      | _ → raise SyntaxError)
  | Rule4 → (function
      | tP :: ts → Rule (Rule4, [tP]) :: ts
      | _ → raise SyntaxError)
  | Rule5 → (function
      | tS :: ts → Rule (Rule5, [tS]) :: ts
      | _ → raise SyntaxError)
  | Rule6 → (function
      | tI :: ts → Rule (Rule6, [tI]) :: ts
      | _ → raise SyntaxError)
  | Rule7 → (function
      | tR :: tE :: tL :: ts → Rule (Rule7, [tL; tE; tR]) :: ts
      | _ → raise SyntaxError)

type configuration = ptree list * input * prediction

let string_of_action = function
  | Recognize s → string_of_symbol s
  | Produce r → string_of_rule r

let rec parse (configuration : configuration) : ptree =
  print_configuration configuration;
  match configuration with
    | (pts, tokens, Produce rule :: prediction) →
      let pts' = apply_rule rule pts in
      parse (pts', tokens, prediction)

    | (pts, token :: input, Recognize (Terminal token') :: prediction) →
      (* Étape d'acceptation d'un caractère. *)
```

```
      if token = token' then
        parse (Leaf token :: pts, input, prediction)
      else
        raise (TokenError (token, token'))

  | (pts, token :: input, Recognize (NonTerminal s) :: prediction) →
    (* Étape de prédiction. *)
    let rule = table s token in
    parse (pts, token :: input, recognize_using rule @ prediction)

  | ([ptree], [], []) →
    ptree

  | _ →
    raise SyntaxError

let parse input = parse ([], input, [Recognize (NonTerminal S)])
```

8. *( ) Comment obtenir un arbre de syntaxe abstraite à partir de l'arbre de production produit par votre analyseur syntaxique ?*
   Réponse:

```
type exp =
  | Add of exp * exp
  | Mul of exp * exp
  | Int

let exp_of_ptree =
  let rec aux = function
    | Leaf INT → (function [] → Int | _ → assert false)
    | Rule (Rule1, [t ; _]) → aux t
    | Rule (Rule2, [tO ; tE ; tE']) → (function [] → (aux tO) [ get tE ; get tE' ] | _ → assert false)
    | Rule (Rule3, [tA]) → aux tA
    | Rule (Rule4, [tP]) → (function [a ; b] → Add (a, b) | _ → assert false)
    | Rule (Rule5, [tS]) → (function [a ; b] → Mul (a, b) | _ → assert false)
    | Rule (Rule6, [tI]) → aux tI
    | Rule (Rule7, [_ ; tE ; _]) → aux tE
    | _ → assert false

  and get i : exp = aux i []

  in
  get
```

□

## Exercice 2

$$S \to E\$ \qquad E \to T + E \mid T!$$
$$T \to F * T \mid F! \qquad F \to n \mid (E)$$

1. *Calculez la fonction FIRST pour tous les non terminaux de cette grammaire.*
   Réponse: FIRST (S) = FIRST (E) = FIRST (T) = FIRST (F) = { n, ( }

2. *La grammaire est-elle LL(1) ? Si oui, quelle est la table d'analyse correspondante ? Si non, existe-t-il une grammaire équivalente LL(1) ? Dans ce cas, donnez-en la table d'analyse.*

   Réponse: Pour savoir si une grammaire est LL(1), il suffit d'essayer de construire la table d'analyse LL(1) de la grammaire. Ainsi, on obtient :

| | $*$ | $+$ | $($ | $)$ | $n$ | $!$ | $\$$ |
|---|---|---|---|---|---|---|---|
| $S$ | $\emptyset$ | $\emptyset$ | $S \to E\$$ | $\emptyset$ | $S \to E$ | $\emptyset$ | $\emptyset$ |
| $E$ | $\emptyset$ | $\emptyset$ | $E \to T+E$<br>$E \to T!$ | $\emptyset$ | $E \to T+E$<br>$E \to T!$ | $\emptyset$ | $\emptyset$ |
| $T$ | $\emptyset$ | $\emptyset$ | $T \to F*E$<br>$T \to F!$ | $\emptyset$ | $T \to F*E$<br>$T \to F!$ | $\emptyset$ | $\emptyset$ |
| $F$ | $\emptyset$ | $\emptyset$ | $F \to (E)$ | $\emptyset$ | $F \to n$ | $\emptyset$ | $\emptyset$ |

   Cette grammaire n'est pas LL(1) car il y a deux règles dans 4 cases. Pour la rendre LL(1), il faut **factoriser à gauche** ces règles en réécrivant la grammaire initiale de la façon suivante :

$$\begin{array}{ll} S \to E\$ & E \to TE' \\ & E' \to +E \mid ! \\ T \to FT' & \\ T' \to *T \mid ! & F \to n \mid (E) \end{array}$$

□

**Exercice 3** *Considérez la grammaire suivante :*

$$\begin{array}{ll} S \to E\$ & E \to E+T \mid E-T \mid T \\ T \to T*F \mid F & F \to id \mid (E) \mid (num) \end{array}$$

1. *Pourquoi cette grammaire n'est pas LL(1) ?*

   Réponse: Il y a au moins deux raisons à cela :
   – $FIRST(E+T) = FIRST(E-T)$ ;
   – aucune grammaire avec récursion gauche est LL(1).

2. *Éliminez les récursions gauches.*

   Réponse: On obtient :

$$\begin{array}{l} S \to E\$ \\ E \to TE' \\ E' \to +TE' \mid -TE' \mid \\ T \to FT' \\ T' \to *FT' \mid \\ F \to id \mid (E) \mid (num) \end{array}$$

3. *Est-ce que la grammaire obtenue ainsi est LL(1) ?*

   Réponse: La grammaire obtenue après élimination n'est pas LL(1) à cause de $F \to (E)$ et $F \to (num)$. Elle est LL(2). On peut la factoriser pour la rendre LL(1) en introduisant les règles : $F \to (F'$ et $F' \to E) \mid num)$

□

**Exercice 4** *On considère la grammaire :*

$$\begin{array}{llll} S & \to E\$ & E & \to TE' \\ E' & \to +TE' \mid & T & \to FT' \\ T' & \to *FT' \mid & F & \to (E) \mid id \end{array}$$

1. *Calculez les symboles annulables (c'est-à-dire les symboles qui ont $\dot{}$ dans $FIRST$)*

   Réponse:
   $E'$ et $T'$ sont annulables.

2. *Calculez FIRST des symboles non terminaux.*
   Réponse:
   $\overline{FIRST}(S) = FIRST(E) = FIRST(T) = FIRST(F) = \quad \}$

Réponse: Oui, et pour faciliter la définition on donne un automate à pile avec acceptation à pile vide. Soit $A = (Q = \{q_a, q_0, q_1, q'_1, q_f\}, \Sigma = \{a, b, 0, 1\}, \Gamma = \{a, Z_0\}, \delta, q_a, Z_0, q_f)$ où les transitions de l'automate sont :

$$\begin{array}{lcllcl} \delta(q_a, aw, \pi) & = & (q_a, a\pi) & \delta(q_a, 0w, \pi) & = & (q_0, \pi) \\ \delta(q_0, bw, a\pi) & = & (q_0, \pi) & \delta(q_a, 1w, \pi) & = & (q_1, \pi) \\ \delta(q_1, bw, a\pi) & = & (q'_1, \pi) & \delta(q'_1, bw, a\pi) & = & (q_1, \pi) \end{array}$$

5. *( ) Pouvez-vous écrire un programme CAML qui reconnaît ce langage ?*
   Réponse:

```
type token = Ta | Tb | T1 | T0 | EOF

exception Unexpected of token

let parse tokens =
  let (accept, next, current) =
    let rec state = ref tokens in
    let next () = match !state with
      | [] → state := [EOF]
      | _ :: toks → state := toks
    in
    let current () = match !state with
      | [] → EOF
      | tok :: _ → tok
    in
    let accept tok =
      if tok ≠ current () then raise (Unexpected (current ()));
      next ()
    in
    (accept, next, current)
  in
  let rec parse_S () =
    parse_A_or_B (fun () → ()) (fun () → ());
    accept EOF

  and parse_A_or_B parse_end_of_A parse_end_of_B =
    match current () with
      | Ta →
        next ();
        parse_A_or_B
          (fun () → accept Tb; parse_end_of_A ())
          (fun () → accept Tb; accept Tb; parse_end_of_B ())
      | Tb →
        raise (Unexpected Tb)
      | T0 →
        next ();
        parse_end_of_A ()
      | T1 →
        next ();
        parse_end_of_B ()
  in
  parse_S ()
```

Une autre solution existe si on utilise un entier pour compter les symboles a lus.

□

**Exercice 6** *On considère la grammaire suivante :*

$$S \to iEtSS' \mid a \qquad S' \to eS \mid \qquad E \to b$$

*Expliquez pourquoi cette grammaire n'est pas LL(1).*
Réponse: Elle n'est pas LL(1) car $e \in First(S') \cap Follow(S')$.

*La grammaire est-elle ambiguë ?*
Réponse: Oui. Par exemple, le mot *ibtibtaea* peut être engendré par deux arbres de dérivation différents.

□

**Exercice 7** *Considérez la grammaire suivante :*

$$\begin{array}{lll} S \to S'\$ & S' \to S'AB \mid S'BC \mid BB & \\ A \to aAa \mid & B \to bB \mid & C \to cC \mid \end{array}$$

*Calculez les symboles annulables et les ensembles FIRST et FOLLOW. Est-ce que la grammaire est LL(1) ?*
Réponse: $A$, $B$ et $C$ sont clairement annulable à cause des règles $A \to$ . Alors $S'$ est annulable à cause de la règle $S' \to BB$.

Les équations pour le calcul de FIRST et FOLLOW sont :

$$\begin{aligned} FIRST(A) &= \{a, \cdot\} \cup FOLLOW(A) \\ FOLLOW(A) &= \{a\} \cup FIRST(B) \cup FOLLOW(B) \\ FIRST(B) &= \{b, \cdot\} \cup FOLLOW(B) \\ FOLLOW(B) &= FIRST(C) \cup FIRST(B) \cup FOLLOW(S') \\ FIRST(S') &= FIRST(S') \cup FIRST(B) \\ FOLLOW(S') &= \{\$\} \cup FIRST(A) \cup FIRST(B) \\ FIRST(S) &= FIRST(S') \cup FOLLOW(S') \end{aligned}$$

En faisant une résolution itérative de ce système en partant de $\emptyset$ on obtient que les ensembles FIRST de $A, B, C, S'$ sont égaux à $\{a, b, c, \$, \cdot\}$ et FISRT$(S)$ et tous les ensembles FOLLOW sont égaux à $\{a, b, c, \$\}$.

La grammaire n'est pas LL(1) car elle est récursive ou, par exemple, $FIRST(A) \cap FOLLOW(A) \neq \emptyset$.

□