Guillaume Kineider

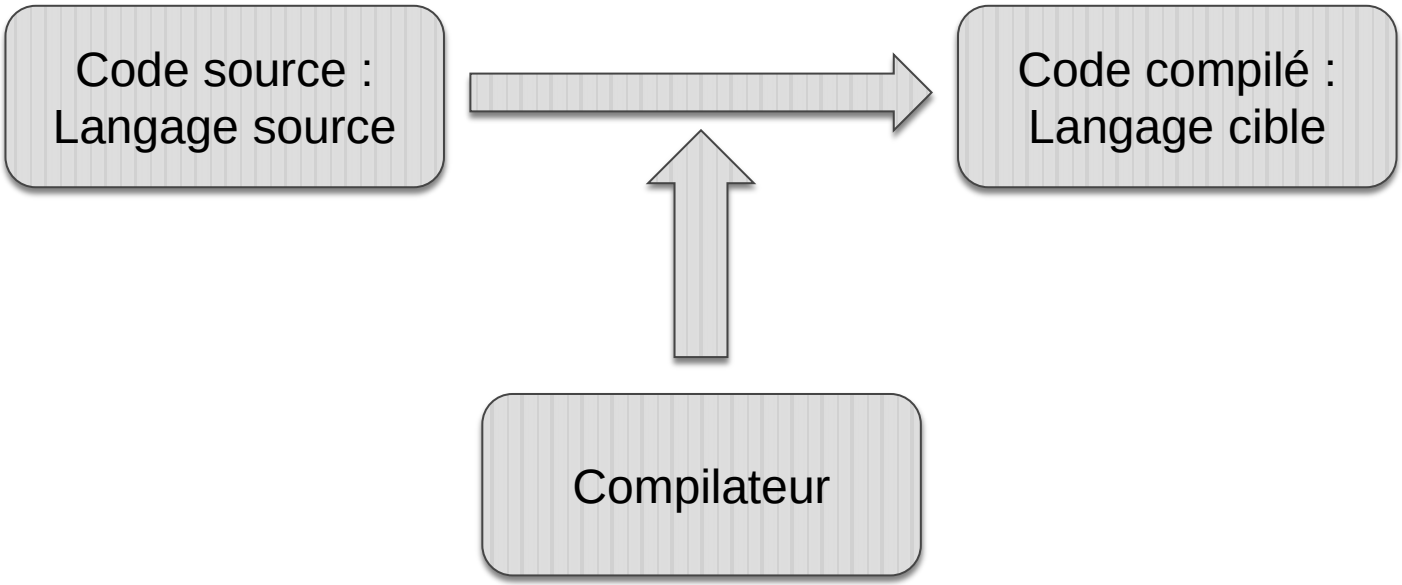
La compilation

Comment fonctionne un compilateur ?

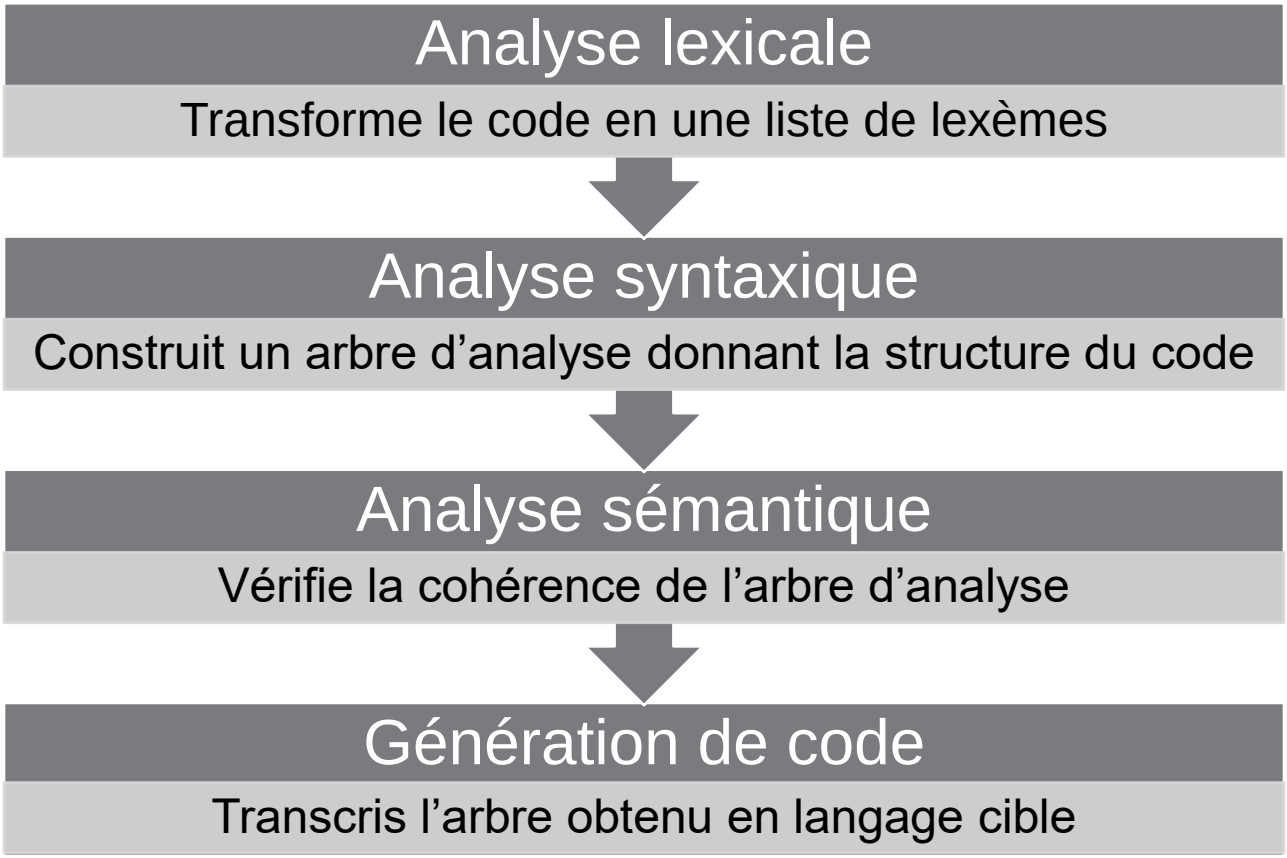
Plan :

- I) Présentation de la compilation
- II) Nos objectifs
  - 1) Présentation des langages utilisés
  - 2) Structures acceptées par le compilateur
- III) La réalisation
  - 1) Utilisation de CamlLex et CamlYacc
  - 2) Analyse lexicale par automate produit
  - 3) Analyse syntaxique par algorithme LL(1)

Principe de base de la compilation :



Etapes de la compilation :



Un exemple :

```
Let var2 = 2 + var1
```

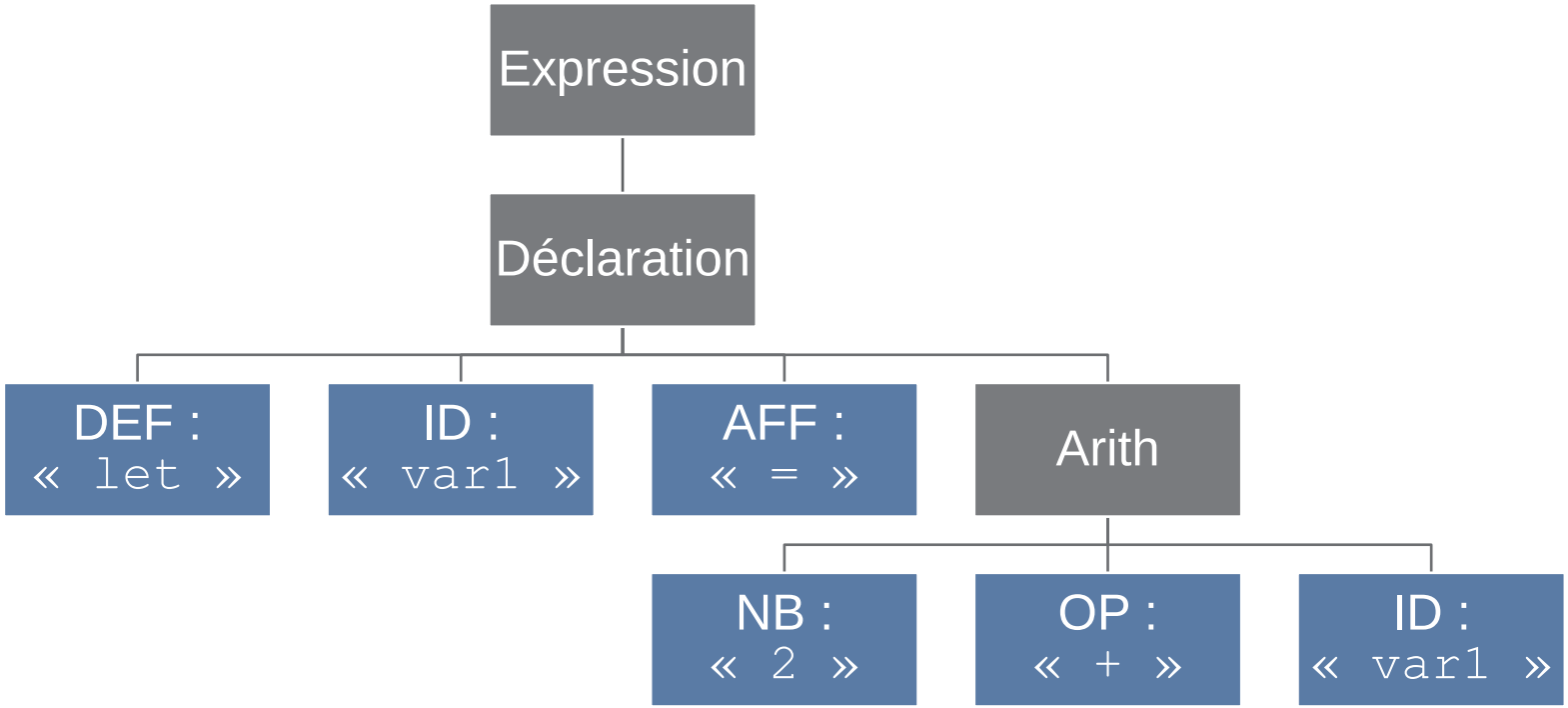
- Analyse lexicale :

DEF	ID	AFF	NB	OP	ID
• « Let »	• « var2 »	• « = »	• « 2 »	• « + »	• « var1 »

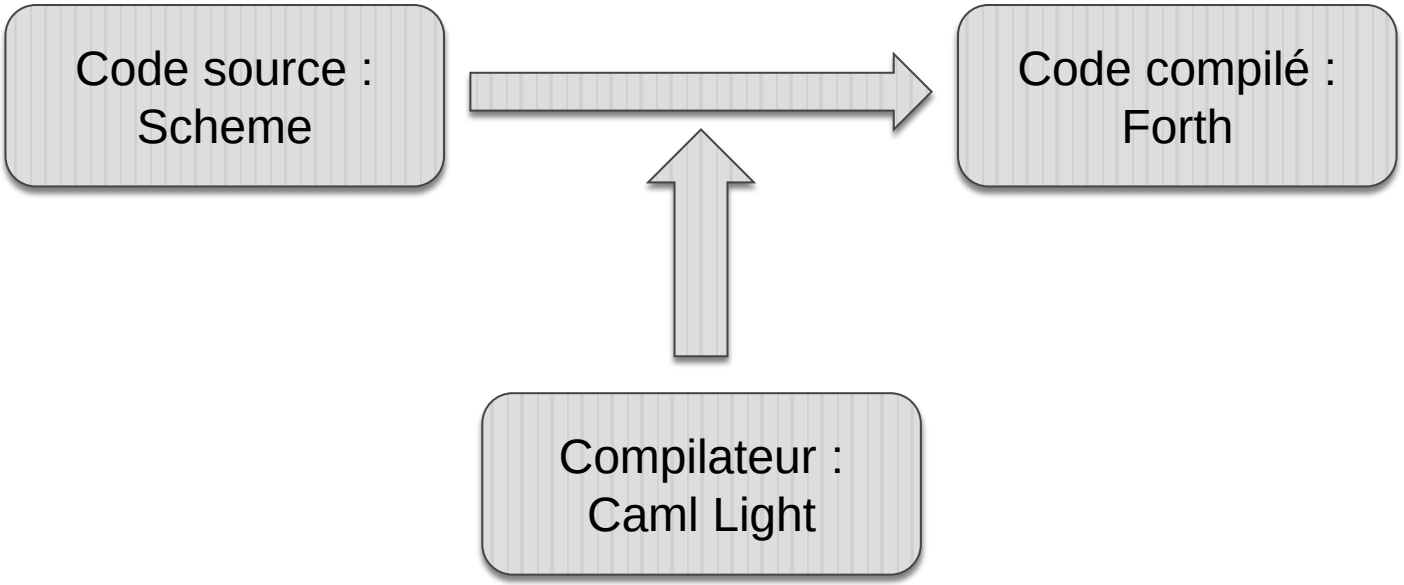
Un exemple :

```
Let var2 = 2 + var1
```

- Analyse syntaxique :



Nos objectifs :



Langages utilisés

- Scheme : notation polonaise
  - Exemple :

```
(define var2 (+ 2 var1))
```

- Forth : notation polonaise inversé avec système de pile
  - Exemple :

```
variable var2 2 var1 + variable2 !
```

Notre Scheme :

- Opérations arithmétiques :
  - `(OP opérande1 opérande2 opérande3 …)`
- Définitions de variables :
  - `(define var1 valeur)`
- Définitions de fonctions :
  - `(define (f arg1 arg2 …) valeur)`
- Application de fonctions :
  - `(f arg1 arg2 …)`

Premier essai : CamlLex et CamlYacc

CamlLex

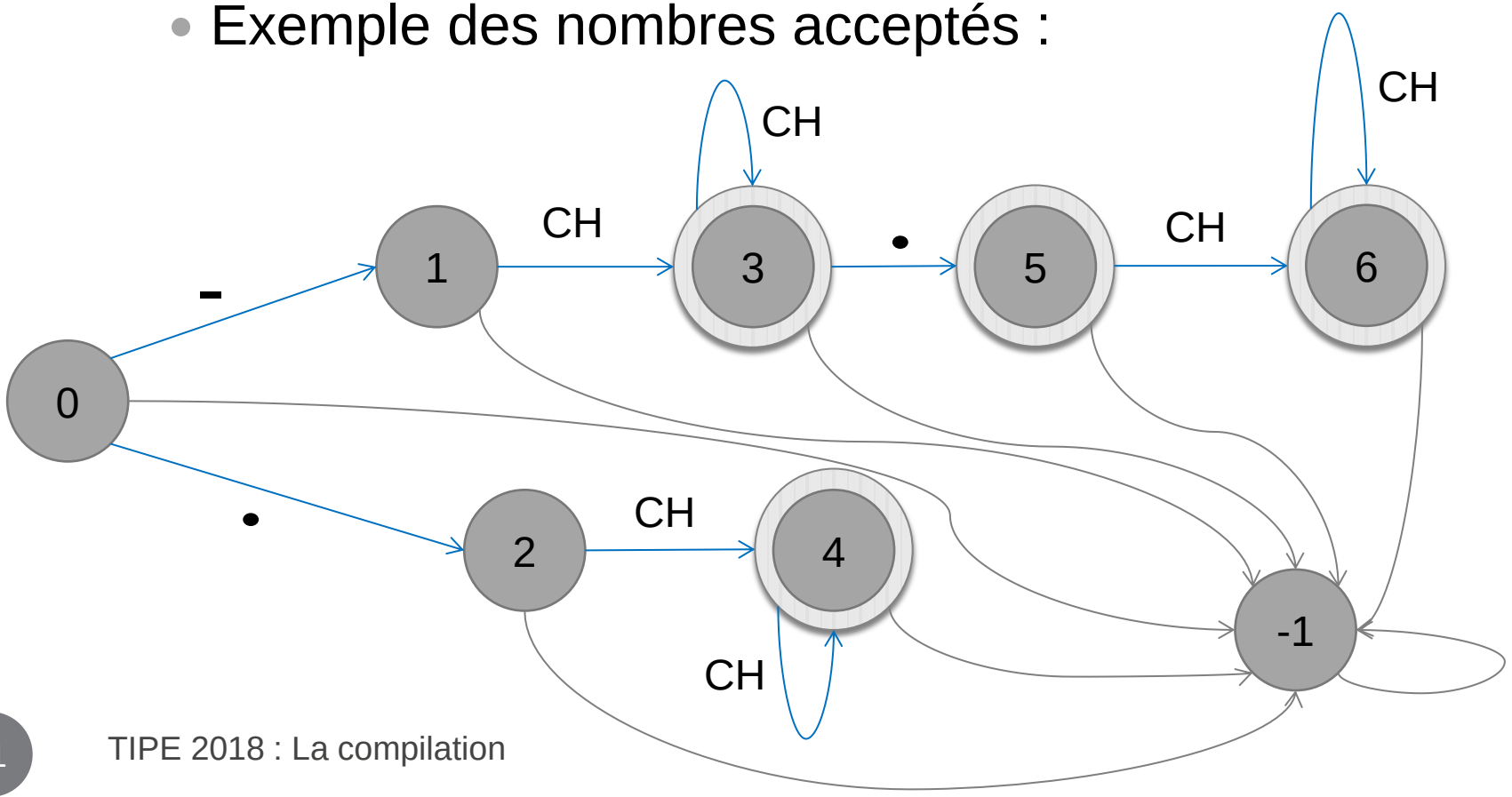
```
(*
Analyse lexicale
 *)

{
  #open "parser";;
  exception EOF;;
}

rule Token = parse
    [` ``\t``\n`]               { Token lexbuf }
  | (`-`?)([`0`-`9`]+)  { NUM(get_lexeme lexbuf) }
  | `+`                         { PLUS }
  | `-`                         { MOINS }
  | `*`                         { FOIS }
  | `/`                         { DIV }
  | `(`                         { LPAR }
  | `)`                         { RPAR }
  | eof                         { raise EOF }
;;
```

CamlYacc

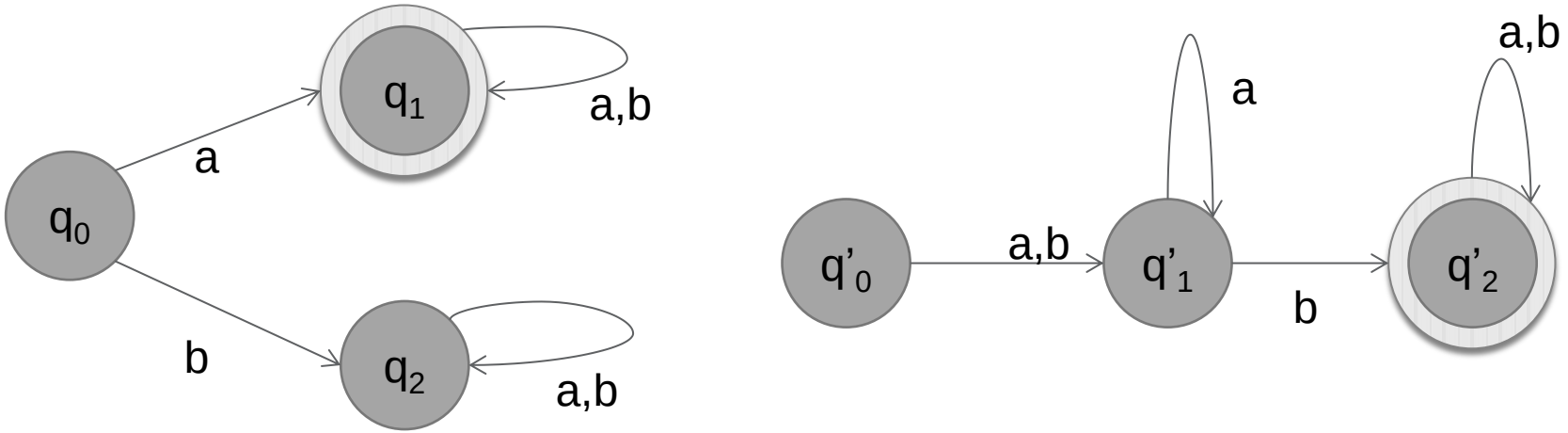
```
/* Analyse Syntaxique */

%token <string> NUM STR
%token PLUS MOINS FOIS DIV
%token LPAR RPAR
%start Main
%type <string> Main
%%

Main :
        LPAR Expr RPAR  { $2 }
;

Expr :
        NUM                             { $1 ^ " " }
  | LPAR Expr RPAR              { $2 }
  | PLUS Expr Expr              { $2 ^ $3 ^ "+ " }
  | MOINS Expr Expr             { $2 ^ $3 ^ "- " }
  | FOIS Expr Expr              { $2 ^ $3 ^ "* " }
  | DIV Expr Expr               { $2 ^ $3 ^ "/ " }
;;
```

Analyse lexicale par automate produit :

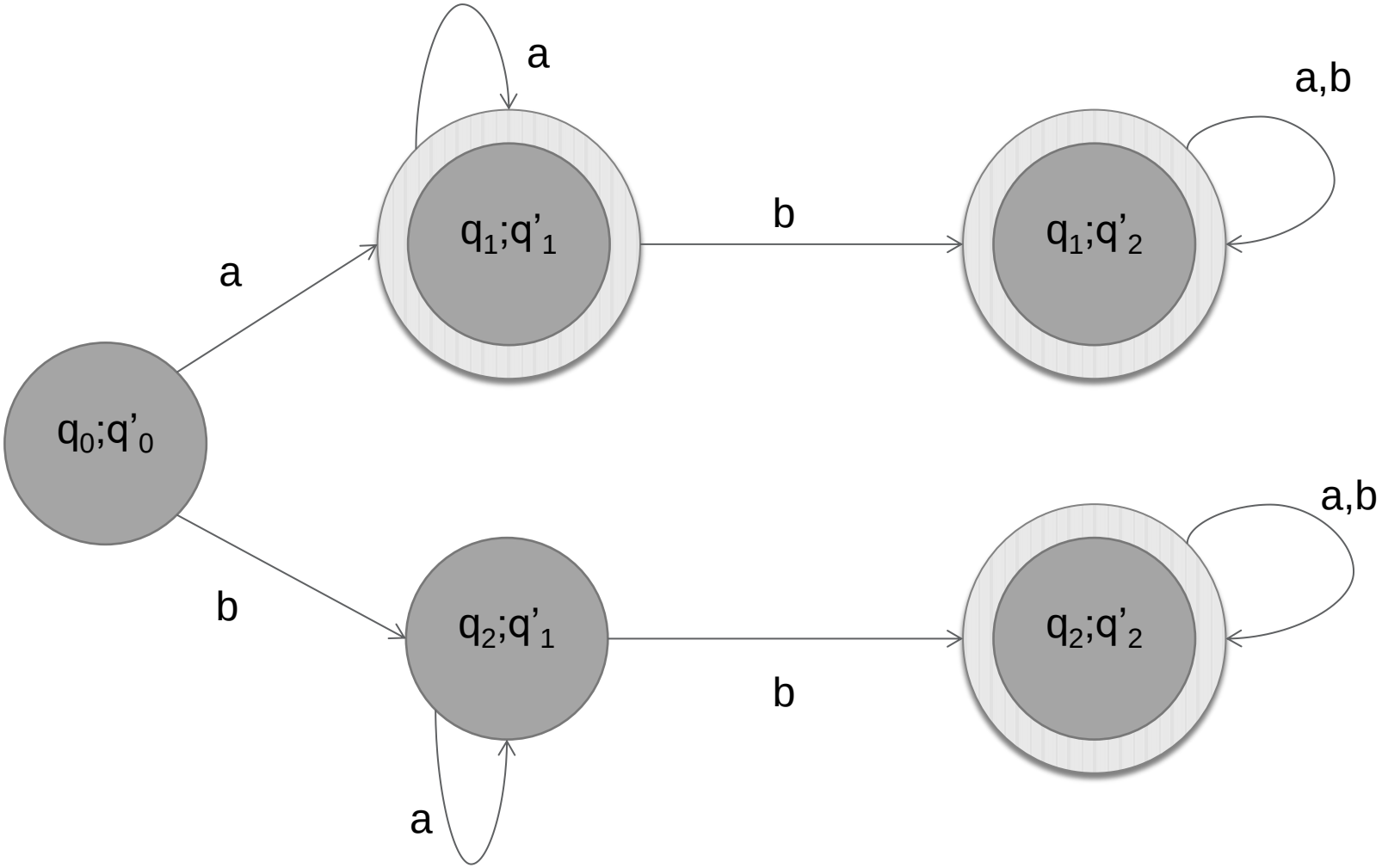
- Description des lexèmes par des expressions rationnelles :
  - Exemple des nombres acceptés :



Analyse lexicale par automate produit :

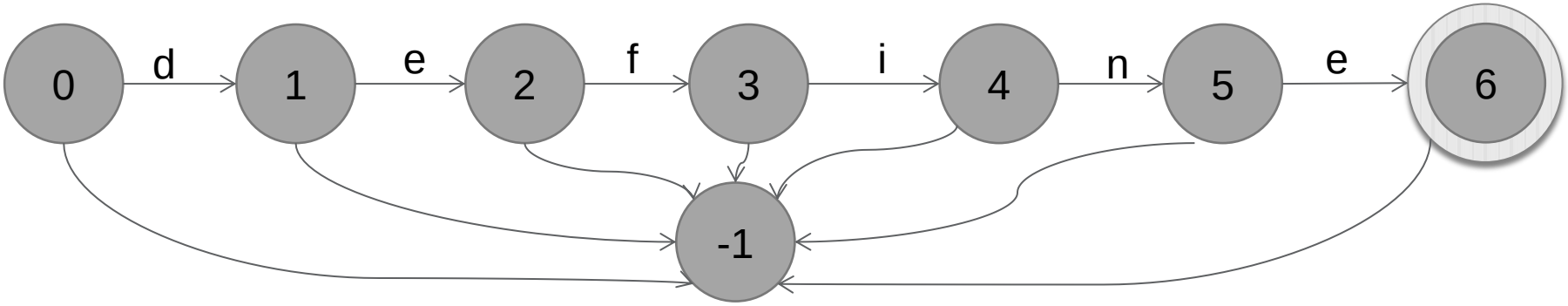


Analyse lexicale par automate produit :

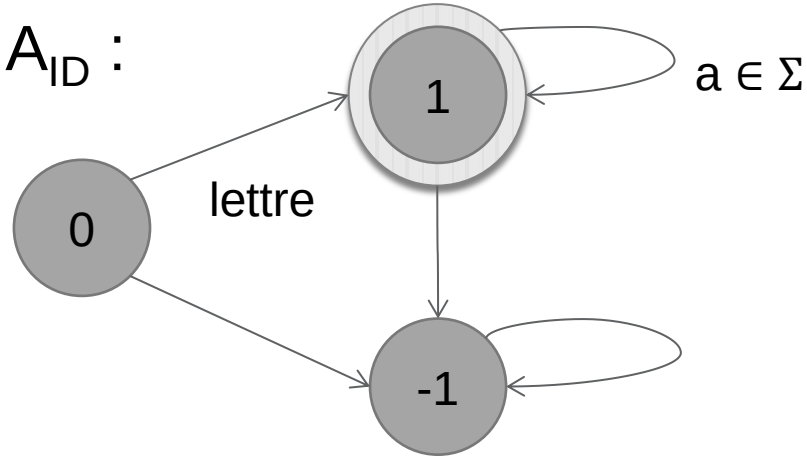


Dans la pratique:

A_{DEF} :

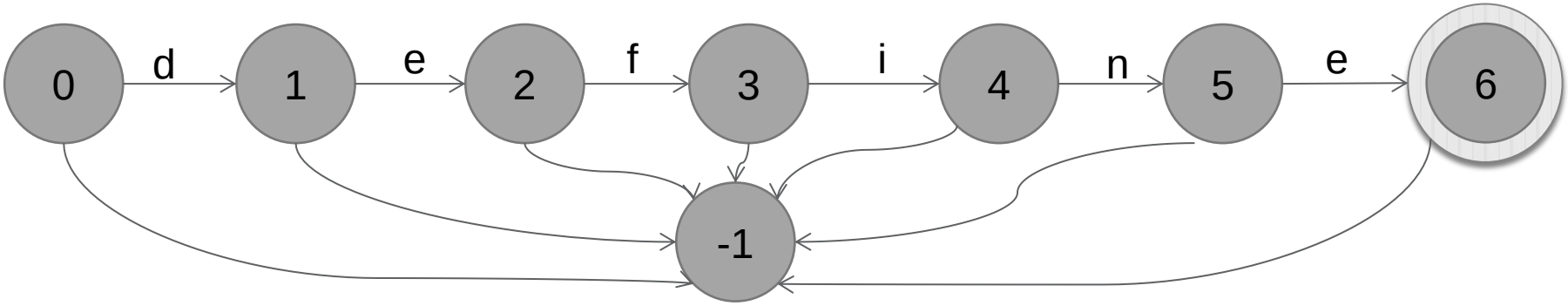


A_{ID} :

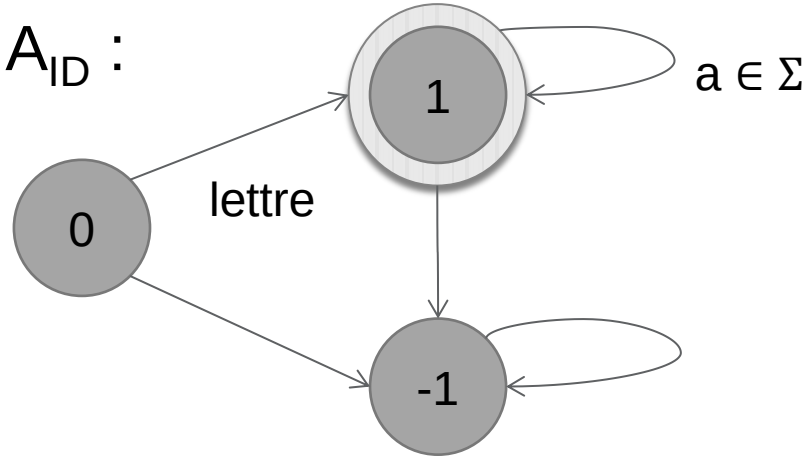


Dans la pratique:

A_{DEF} :



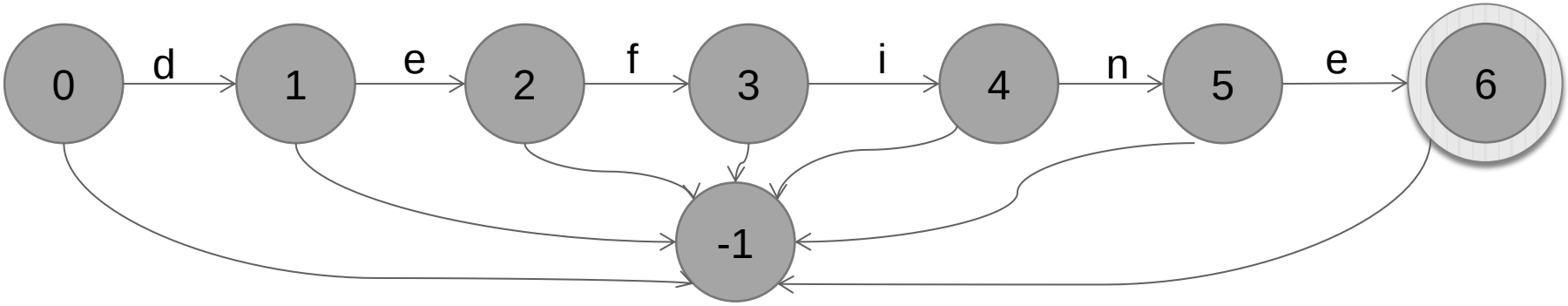
A_{ID} :



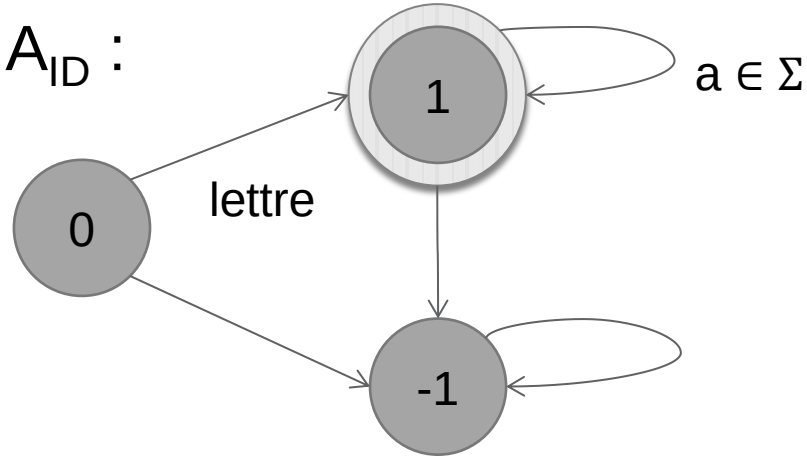
Lettre	A_{DEF}	A_{ID}	A_{NB}
-	0	0	0

Dans la pratique:

A_{DEF} :



A_{ID} :



Lettre	A_{DEF}	A_{ID}	A_{NB}
-	0	0	0
d	1	1	-1
e	2	1	-1
f	3	1	-1
i	4	1	-1
n	5	1	-1
e	6	1	-1

Résultat de l'analyse lexicale :

```
(define var2 (+ 2 var1))
```

LPAR	DEF	ID	LPAR	OP	NB
• « (»	• « define »	• « var2 »	• « (»	• « + »	• « 2 »

ID	RPAR	RPAR
• « var1 »	• «) »	• «) »

Analyse syntaxique : grammaire

- Une grammaire formelle (ou, simplement, grammaire) est constituée des quatre objets suivants:
  - Un ensemble fini de symboles, appelés *symboles terminaux* (qui sont les « lettres » du langage), notés conventionnellement par des minuscules,
  - Un ensemble fini de symboles, appelés *non-terminaux*, notés conventionnellement par des majuscules,
  - Un élément de l'ensemble des non-terminaux, appelé *axiome*, noté conventionnellement S,
  - Un ensemble de *règles de production*, qui sont des paires formées d'un non-terminal et d'une suite de terminaux et de non-terminaux ; par exemple, $A \rightarrow ABa$.

Analyse syntaxique : grammaire

- `Term={EOF; EPS; LPAR; RPAR; DEF; ID; NB; OP}`
- `Non_term = {ListExpr; Expr; Instr; Var; Arg; Arith; ResteArg; App}`
- `Axiome = ListExpr`

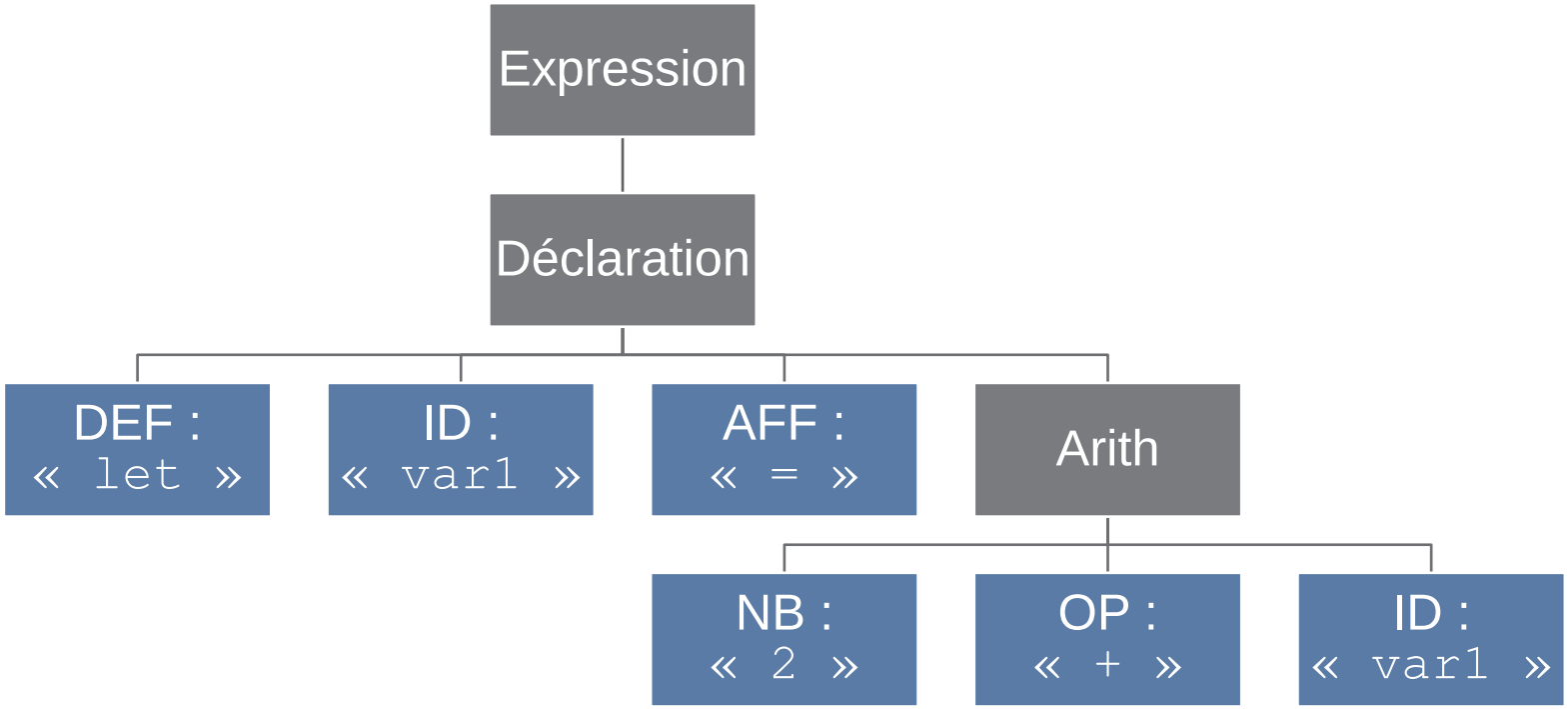
- Règles de production :

```
ListExpr → Expr ListExpr | EPS
Expr → NB | ID | LPAR Instr RPAR
Instr → DEF Var Arith | App
Var → ID | LPAR ID Arg RPAR
Arg → EPS | ID Arg
Arith → NB | ID | LPAR  App RPAR
ResteArg → EPS | Arith ResteArg
App → OP Arith Arith ResteArg | ID ResteArg
```

Analyse syntaxique : Algorithme naïf

`Let var2 = 2 + var1`

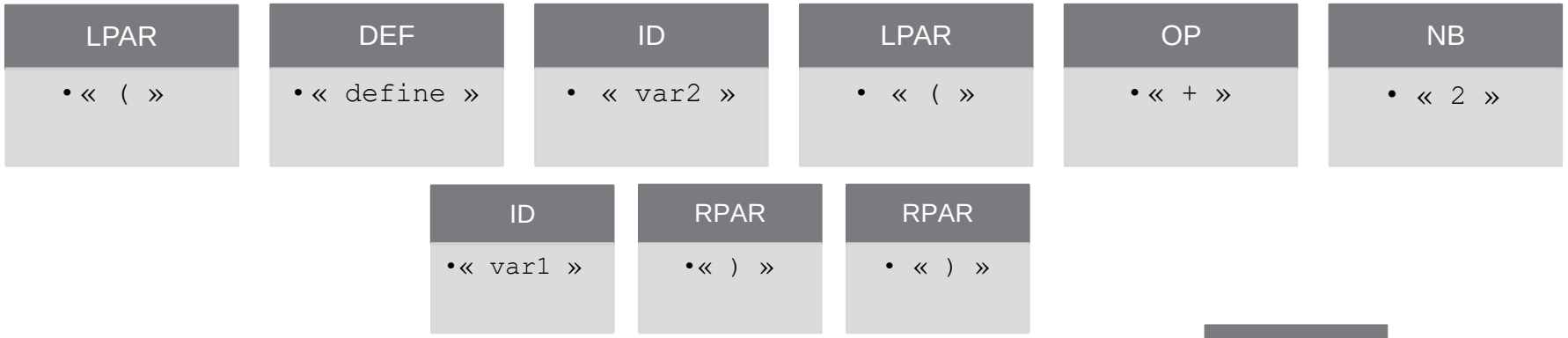
- Arbre syntaxique :



Analyse syntaxique : Algorithme naïf

- On connait le nœud courant et le lexème en entrée.
- On détermine le choix du non-terminal actuel comme étant le seul qui commence par le terminal en entrée.

Exemple :



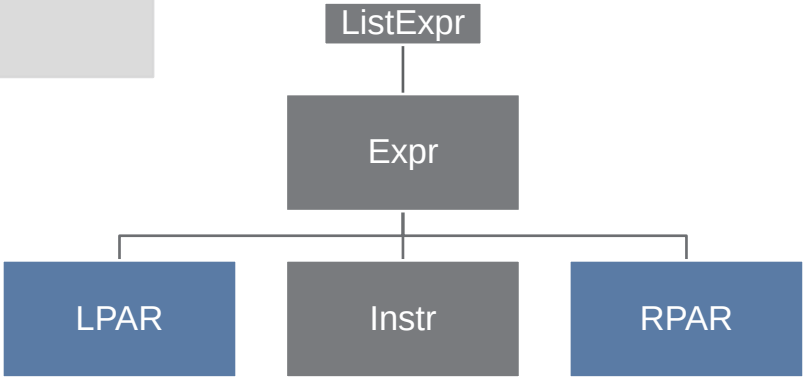
Règles de production :

```
ListExpr → Expr ListExpr | EPS
Expr → NB | ID | LPAR Instr RPAR
Instr → DEF Var Arith | App
Var → ID | LPAR ID Arg RPAR
Arg → EPS | ID Arg
Arith → NB | ID | LPAR  App RPAR
ResteArg → EPS | Arith ResteArg
App → OP Arith Arith ResteArg | ID ResteArg
```

Exemple :

DEF	ID	LPAR	OP	NB
• « define »	• « var2 »	• « (»	• « + »	• « 2 »

ID	RPAR	RPAR
• « var1 »	• «) »	• «) »



Règles de production :

```
ListExpr → Expr ListExpr | EPS
Expr → NB | ID | LPAR Instr RPAR
Instr → DEF Var Arith | App
Var → ID | LPAR ID Arg RPAR
Arg → EPS | ID Arg
Arith → NB | ID | LPAR  App RPAR
ResteArg → EPS | Arith ResteArg
App → OP Arith Arith ResteArg | ID ResteArg
```

Exemple :

ID	LPAR	OP	NB
• « var2 »	• « (»	• « + »	• « 2 »

ID	RPAR	RPAR
• « var1 »	• «) »	• «) »

Règles de production :

```
ListExpr → Expr ListExpr | EPS
Expr → NB | ID | LPAR Instr RPAR
Instr → DEF Var Arith | App
Var → ID | LPAR ID Arg RPAR
Arg → EPS | ID Arg
Arith → NB | ID | LPAR  App RPAR
ResteArg → EPS | Arith ResteArg
App → OP Arith Arith ResteArg | ID ResteArg
```