Augmentons la grammaire : E’ -> E$

E -> id
E -> E (L)
Récursive à gauche
Eliminons cette récursivité :
E -> id X
X -> (L) X
X -> /\

Calculons les symboles directeurs à l’ordre 1 (k = 1)
E’ -> E$
E -> id X
X -> (L) X
X -> /\
L -> E , L
L -> E

SD_1(X -> alpha) = P_1(alpha) \ { /\ } U S_1(X) si /\ in P_1(alpha)

a. P_1(/\) = { /\ }
b. P_1(a beta) = { a } avec a in A
c. P_1(Y beta) = union pour Y -> gamma_i (P_1(gamma_i) \ { /\ } u P_1(beta) \ { /\ } si /\ in P_1(gamma_i))

SD(E’ -> E $) : = { id }
P(E $) = { id }
Calcul par la règle c.
E -> id X : P(id X) = { id } par la règle b

SD(E -> id X) : = { id }
P(id X) = { id } par la règle b

SD(X -> (L) X) : = { (}
P((L)X) = { (} par la règle b

SD(X -> /\ ) : /\ appartient aux Premiers donc SD(X -> /\ ) = P(/\ ) \ { /\ } u S(X) = { $ }
P(/\ ) = { /\ } par la règle a

Toutes les règles contenant X : Y_i -> alpha_i X beta_i
S(X) = P(beta_i) \ { /\ } u S(Y_i) si /\ in P(beta_i)

S(X) :
Règles contenant X :

E -> id X : beta_i = /\
S(X) contiennent P(/\ ) \ { /\ } u S(E)

X -> (L) X : beta_i = /\
S(X) contiennent P(/\ ) \ { /\ } u S(X) : pas de contribution

S(E) :
Règles contenant E :
E’ -> E $: beta_i = $
S(E) contiennent P($) = { $ }

S(E) = { $ }
S(X) = { $ }

SD(L -> E , L) = P(E , L) \ { /\ } u S(L) si /\ in P(E , L) = { id }
SD(L -> E) = P(E) \ { /\ } u S(L) si /\ in P(E) = { id }
Constater le conflit car SD(L -> E , L) intersection SD(L -> E) = { id }
Factoriser les règles
L -> E Y
Y -> , L
Y -> /\
Calculer les symboles directeurs des nouvelles règles
SD(L -> E Y) = P(E Y) \ { /\ } u S(L) si /\ in P(E Y) = { id }
SD(Y -> , L) = { , }
SD(Y -> /\ ) = S(Y) = S(L) = S(Y) u {) } = {) }

Construction d’un analyseur descendant récursif :

Type pour les unités lexicales :

```
type token = PAROUV | PARFER | VIRG | ID of  string | DOLLAR;;
type word = token list;;

let rec parseEprime w =
match w with
(ID text) :: wp -> (* SD( E’ -> E $ *)
        (match (parseE  w) with
        Success wpp ->
                (match wpp with
                DOLLAR -> Success []
                | _ -> ParserError)
| _ -> ParseError

and parseE w =
match w with
```

```
(ID text) :: wp -> parseX wp (* SD(E -> id X *)
| [DOLLAR] -> Success w
| _ -> ParseError

and parseX w =
match w with
PAROUV :: wp -> (* SD( X -> ( L ) X *)
        (match (parseL wp) with
        PARFER :: wpp -> parseX wpp
        | _ -> parseError)
| [DOLLAR] -> Success w
| _ -> ParseError

and parseL w =

and parseY w =
```