⇨ **Grammaire de base :**

<file> ::= NEWLINE? <def>* <stmt>+ EOF

<def> ::= def <ident> (<ident>*,) : <suite>

<suite> ::= <simple_stmt> NEWLINE

| NEWLINE BEGIN <stmt>+ END

<simple_stmt> ::= return <expr>

| <ident> = <expr>

| <expr> [<expr>] = <expr>

| print (<expr>)

| <expr>

<stmt> ::= <simple_stmt> NEWLINE

| if <expr> : <suite>

| if <expr> : <suite> else : <suite>

| for <ident> in <expr> : <suite>

<expr> ::= <const> | <ident> | <expr> [<expr>] | - <expr> | not <expr> | <expr> <binop> <expr> | <ident> (<expr>*,) | [<expr>*,] | (<expr>)

<binop> ::= + | - | * | // | % | <= | >= | > | < | != | == | and | or

<const> ::= <integer> | <string> | True | False | None

⇨ **Introduction des priorités des opérateurs**

<expr> := or_expr

<or_expr> := <and_expr>$^{+}_{or}$
<and_expr> := <not_expr>$^{+}_{and}$
<not_expr> := not? <comp_expr>
<comp_expr> := <add_expr> (<binop_comp> add_expr) ?
<add_expr> := <mut_expr>$^{+}_{<binop\_add>}$
<mut_expr> := <minus_expr>$^{+}_{<binop\_mut>}$

<minus_expr> := - ? <crochet_expr>

<crochet_expr> := <terminal_expr> ([<expr>]) *

<terminal_expr> := <const> | <ident>
| <ident> (<expr>*,)
| [<expr>*,] | (<expr>)

<binop_comp> := <= | >= | > | < | != | ==
<binop_add> := + | -
<binop_mut> := * | // | %

⇨ **Explicitation des règles * + et ? avec ou sans symbole :**

<exemple>+	<exemple_plus> := <exemple> <exemple_plus_rest> <exemple_plus_rest> := ε | <exemple_plus>
<exemple>*	<exemple_etoile> -> ε | <exemple> <exemple_etoile>
<exemple>+$_{symbole}$	<exemple_plus_symbole> := <exemple> <exemple_plus_symbole_rest> <exemple_plus_symbole_rest> := ε | symbole <exemple_plus_symbole>
<exemple>*$_{symbole}$	<exemple_etoile_symbole> := ε | <exemple_plus_symbole>
exemple?	<exemple> := ε | exemple*

⇨ **Factorisation des règles :**

<stmt> := if <expr> : <suite> <stmt> := if <expr> : <suite> else : <suite>	<stmt> := if <expr> : <suite> <stmt_else> <stmt_else> := ε | else : <suite>
<terminal_expr> := ... | <ident> | <ident> (<expr>*,) | ...	<terminal_expr> := <ident> <ident_fact> <ident_fact> := ε | (<expr>*,)

⇨ **Ambiguïté avec <simple_stmt> :**

On a :
<simple_stmt> ::= ... | <expr> [<expr>] = <expr>

| <expr> | <ident> = <expr>

Quelle regle choisir quand on lit "<ident>" ou "<expr> [" ?

Le plus simple serait de remplacer <expr> [<expr>] et <ident> par <expr> :

→ Probleme : on elargie la grammaire. On peut maintenant ecrire <string> = <integer> par exemple

Supposons que ce probleme n'en est pas un, on a alors

- <simple_stmt> ::= ... | <expr> = <expr>| <expr> | ....

On factorise

- <simple_stmt> ::= ... | <expr> <simple_stmt_fact> | ....

  <simple_stmt_fact> -> ε | = <expr>

La grammaire devient alors LL1

Revenons sur notre probleme. Il n'y a pas de restrictions s'il y a juste <expr>, mais il y en a s'il s'agit d'une affectation <expr> = <expr>, auquel cas le membre gauche ne peut être qu'un identificateur, ou un accès via indice.

Il faudrait donc pouvoir regarder après l'analyse du premier <expr>

Or une implémentation tel que : une règle == une méthode/fonction, permet un "stockage de cette information"
En effet, on lance l'analyse de la première expression en appelant analyseExpr(). Quand cette méthode termine, on revient dans le corps de la fonction appelante.
Alors on a une disjonction de cas sur le prochain token :

- Si le prochain token est un =, alors
  - si l'expression analysé est un accès par indice ou un identificateur, alors pas de probleme,
  - sinon renvoie d'une erreur
- Sinon retourner l'expression analysé

⇨ **Grammaire finale (syntaxe gramophone) :**

file -> opt_newline def_etoile stmt_plus EOF .

opt_newline -> . opt_newline -> NEWLINE .

def_etoile -> . def_etoile -> deft def_etoile .

stmt_plus -> stmt stmt_plus_rest . stmt_plus_rest -> .

stmt_plus_rest -> stmt_plus .

deft -> def ident "(" ident_etoile_virgule ")" ":" suite .

ident_etoile_virgule -> . ident_etoile_virgule -> ident_plus_virgule . ident_plus_virgule -> ident ident_plus_virgule_rest . ident_plus_virgule_rest -> . ident_plus_virgule_rest -> "," ident_plus_virgule .

suite -> simple_stmt NEWLINE . suite -> NEWLINE BEGIN stmt_plus END .

simple_stmt -> return expr . simple_stmt -> print "(" expr ")" . simple_stmt -> expr simple_stmt_fact .

simple_stmt_fact -> .

simple_stmt_fact -> "=" expr .

stmt -> simple_stmt NEWLINE . stmt -> for ident in expr ":" suite . stmt -> if expr ":" suite stmt_else . stmt_else -> . stmt_else -> else ":" suite .

expr -> or_expr .

or_expr -> and_expr or_expr_rest .

or_expr_rest -> .

or_expr_rest -> binop_or or_expr .

and_expr -> not_expr and_expr_rest .

and_expr_rest -> .

and_expr_rest -> binop_and and_expr .

not_expr -> comp_expr . not_expr -> not comp_expr .

comp_expr -> add_expr comp_expr_rest .

comp_expr_rest -> .

comp_expr_rest -> binop_comp add_expr .

add_expr -> mut_expr add_expr_rest .

add_expr_rest -> .

add_expr_rest -> binop_add add_expr .

mut_expr -> minus_expr mut_expr_rest .

mut_expr_rest -> . mut_expr_rest -> binop_mut mut_expr .

minus_expr -> "-" crochet_expr .

minus_expr -> crochet_expr .

crochet_expr -> terminal_expr expr_crochet_etoile .

expr_crochet_etoile -> .

expr_crochet_etoile -> "[" expr "]" expr_crochet_etoile .

terminal_expr -> "(" expr ")" . terminal_expr -> const . terminal_expr -> ident expr_rest_ident . terminal_expr -> "[" expr_etoile_virgule "]" .

expr_rest_ident -> "(" expr_etoile_virgule ")" . expr_rest_ident -> .

expr_etoile_virgule -> . expr_etoile_virgule -> expr_plus_virgule .

expr_plus_virgule -> expr expr_plus_virgule_rest . expr_plus_virgule_rest -> . expr_plus_virgule_rest -> "," expr_plus_virgule .

binop_add -> "+" . binop_add -> "-" . binop_mut -> "*" . binop_mut -> "//" . binop_mut -> "%" . binop_comp -> "<=" . binop_comp -> ">=" . binop_comp -> ">" . binop_comp -> "<" . binop_comp -> "!=" . binop_comp -> "==" . binop_and -> and . binop_or -> or .

const -> integer . const -> string . const -> True . const -> False . const -> None .

MAIS la grammaire ainsi crée est devenu associative droite et ne respecte donc pas l'associativité gauche.