**Rapport de projet en Techniques de Compilation**

**Fait par :**

**IMLOUL Fatima Zahrae**

**EL HADRAOUI Oumaima**

**Titre du Projet :**

**Analyse de Séquences à l'aide de Grammaire EBNF**

**Plan**

**Résumé ………………………………………………………………….3**

**Introduction …………………………………………………………….3**

**Contexte Théorique …………………………………………………...3**

**Conception et Méthodologique……………………………………...3**

1. **Analyseur lexical (Lexer) ……………………………………….3**
2. **Analyseur syntaxique (Parser)…………………………………7**

**-Les règles de la grammaire EBNF avec leurs automates associés ……………………………………………………………7**

**-Analyse lexicale avec le Lexer ………………………………19**

**-Analyse syntaxique avec le Parser …………………………20**

**Conclusion……………………………………………………………...21**

**Résumé :**

Le projet vise à développer un programme **d'analyse de séquences** en utilisant la grammaire sous forme de règles **EBNF**. En appliquant ces règles, le programme sera capable d'effectuer **une analyse syntaxique** des séquences en entrée, contribuant ainsi à la compréhension des principes fondamentaux de la compilation.

**Introduction :**

L'analyse syntaxique est une étape cruciale dans le processus de compilation, permettant de vérifier la structure grammaticale d'un programme. Ce projet se concentre sur l'utilisation de **la grammaire EBNF** pour décrire formellement **la syntaxe des séquences**. Le développement d'un programme dédié à cette tâche contribuera à renforcer les connaissances pratiques dans le domaine de la compilation.

**Contexte Théorique :**

La **grammaire EBNF** est une extension de la notation de **B**ackus-**N**aur, largement utilisée pour décrire la syntaxe des langages de programmation. Elle utilise des règles de production pour spécifier les constructions grammaticales autorisées. **L'analyse syntaxique** consiste à déterminer si une séquence de **tokens** suit les règles définies par la grammaire.

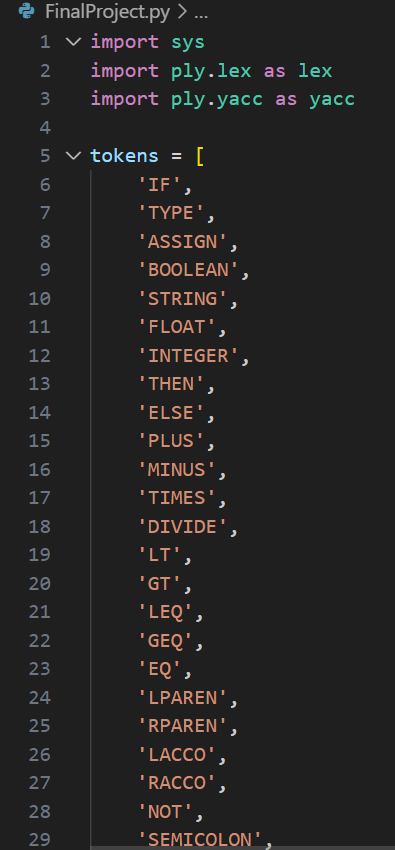
**Conception et Méthodologie**

1. **Analyseur lexical (Lexer) :**

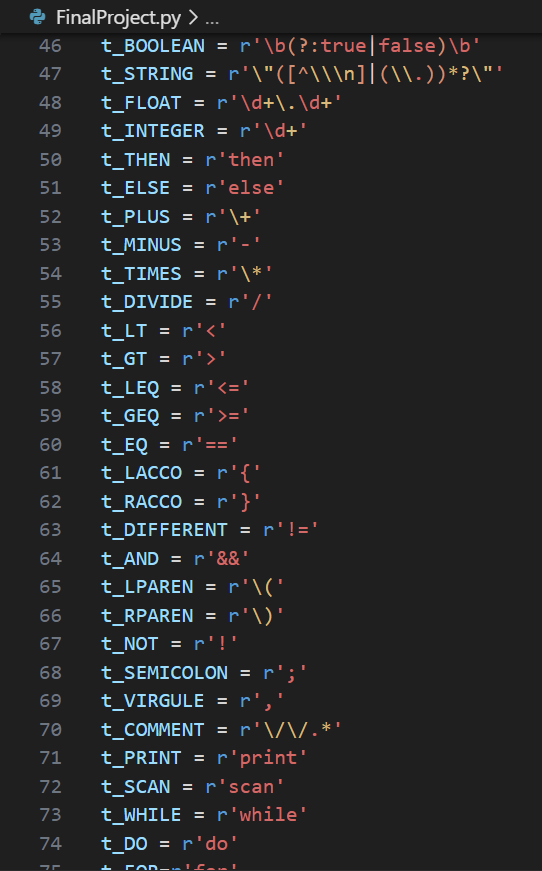
Définitions des tokens et des expressions régulières :

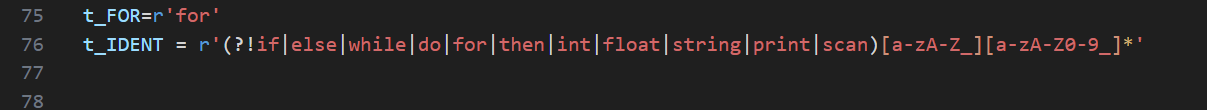
On a défini des tokens tels que **IF**, **TYPE**, **ASSIGN,…,** avec des expressions régulières correspondantes. Ces dernières couvrent divers éléments, y compris les opérations arithmétiques, les opérateurs de comparaison, les mots-clés, …

3



4





Voici quelques tokens avec leurs automates :

IF = ” if ”

THEN = ”then”

ELSE = ”else”

WHILE = “while”

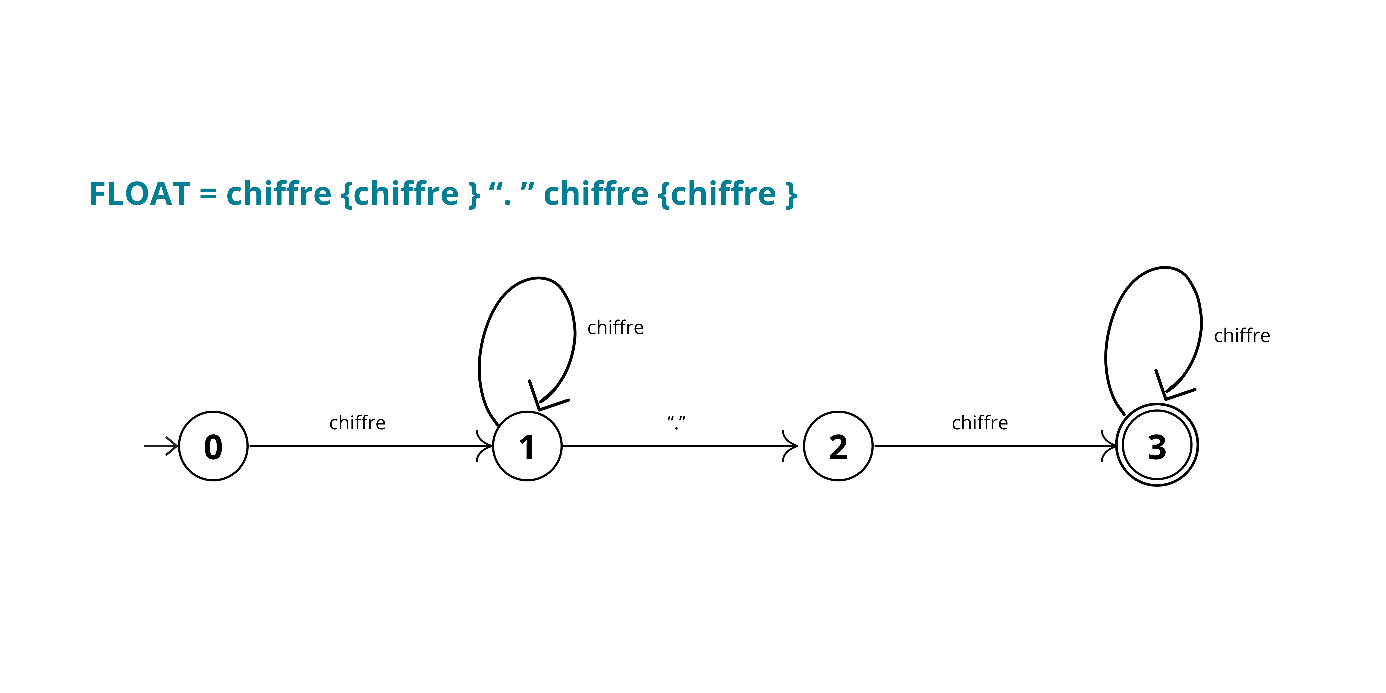
TYPE = “bool” | “string“ | “float“ | “int”

ASSIGN = ”= ”

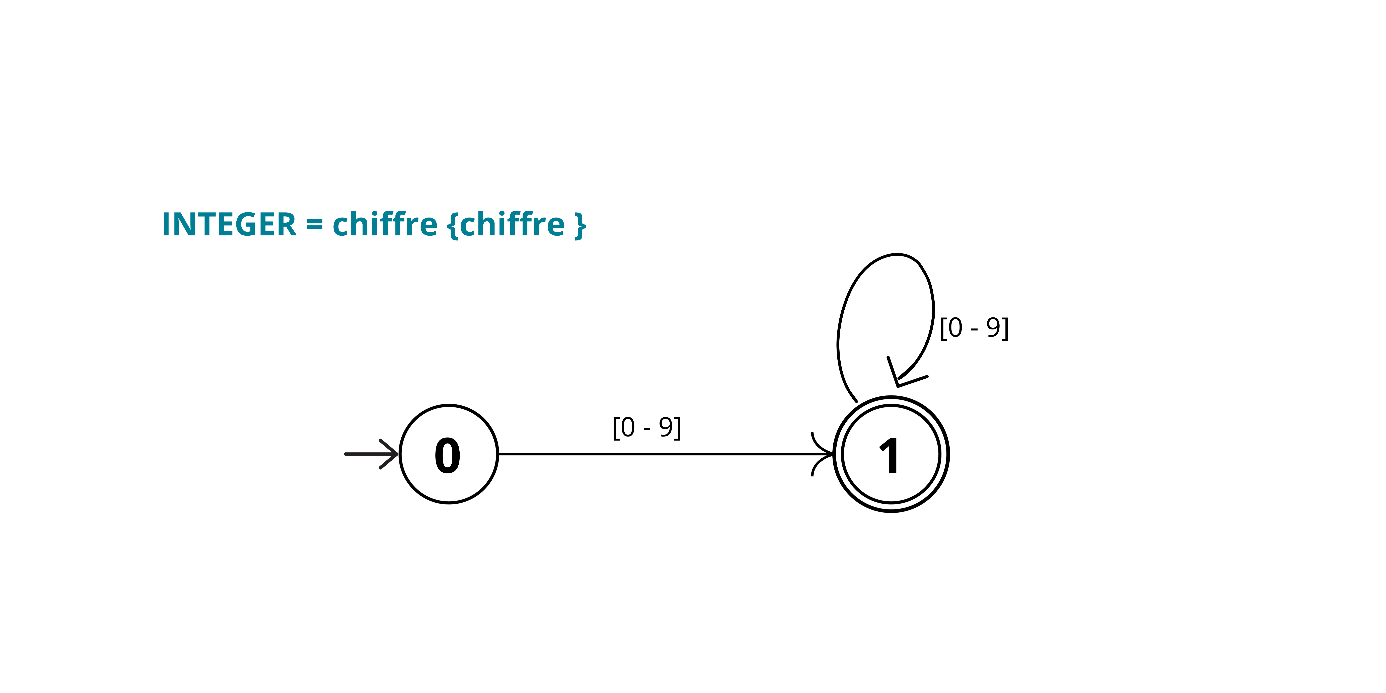
BOOLEAN = ”true” | ”false”

FLOAT = chiffre {chiffre} ”. ” chiffre {chiffre}

5



INTEGER = chiffre {chiffre}



Chiffre=0|1|2|3|4|5|6|7|8|9

PLUS = ”+ ”

AND = ”&& ”

LPAREN = “(”

RPAREN = ”) ”

VIRGULE = “,”

MINUS = “-”

TIMES = “\*”

DIVIDE = “/”

LT = ”< ”

GT = “>”

NOT = ”! ”

PRINT = ”print”

DO = ”do”

FOR = ”for”

LEQ = ”=< ”

GEQ = ”> ”

EQ = ”== ”

LACCO = ”{ ”

RACC = ”} ”

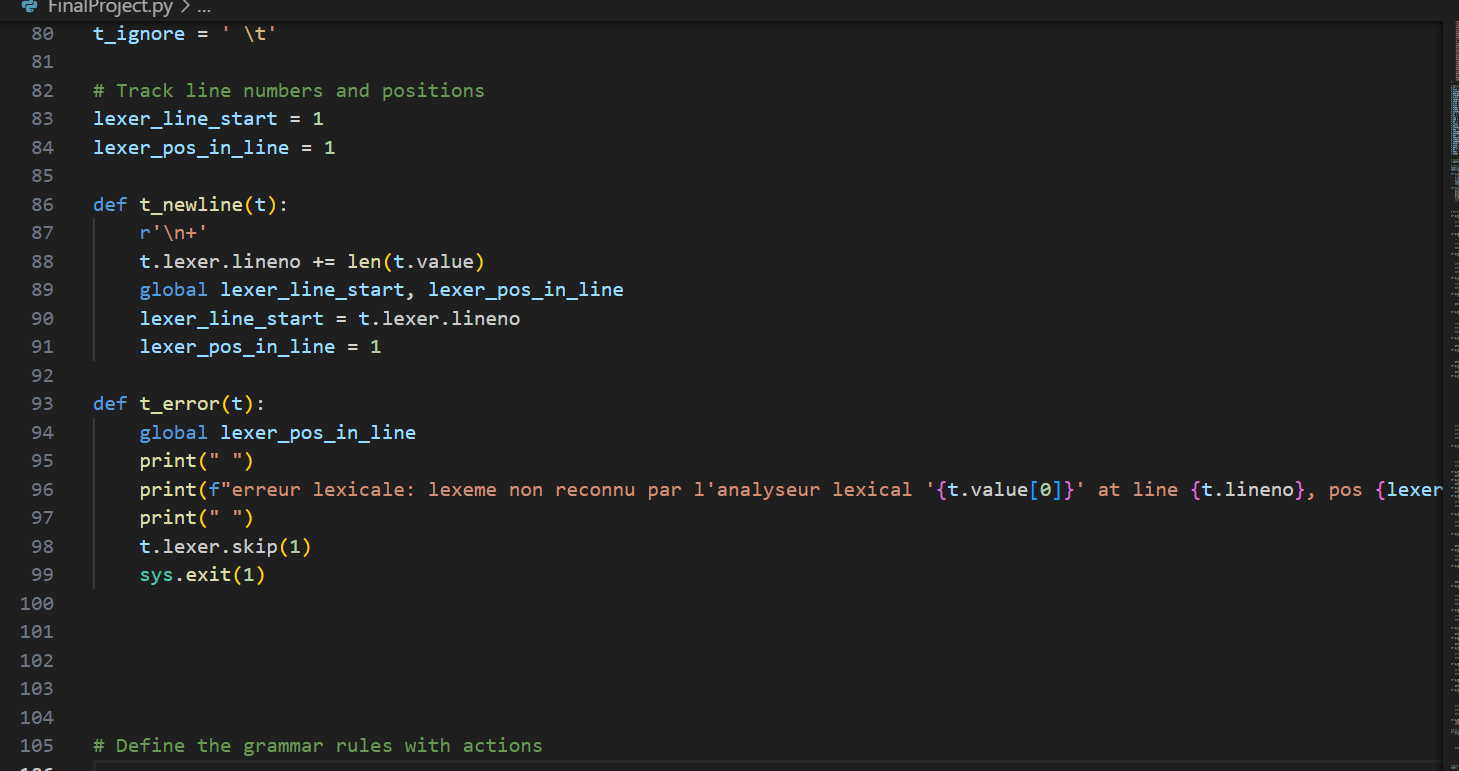
DIFFERENT = ”!= ”

SEMICOLON = ”, ”

IDENT = \*lettre {lettre | chiffre | \_ }

lettre = a | b | ……. | z | A | ……………….. | Z

6



**Caractères Ignorés :** Les espaces et les tabulations sont ignorés (t\_ignore = ' \t').

**Gestion des Nouvelles Lignes** : La fonction t\_newline est définie pour suivre les numéros de ligne lors de la rencontre de caractères de nouvelle ligne.

**Gestion des erreurs lexicales :** La fonction t\_error est définie pour afficher un message d'erreur pour les caractères illégaux.

1. **Analyseur syntaxique (Parser) :**

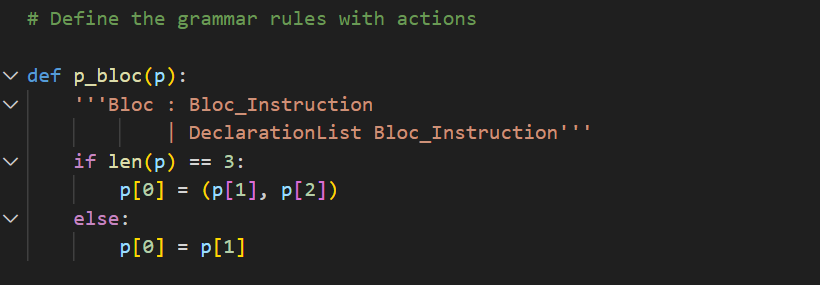
Règles de grammaire :

- Définition **des règles de grammaire** pour différentes constructions syntaxiques, telles que les déclarations, les instructions conditionnelles, les boucles, etc.

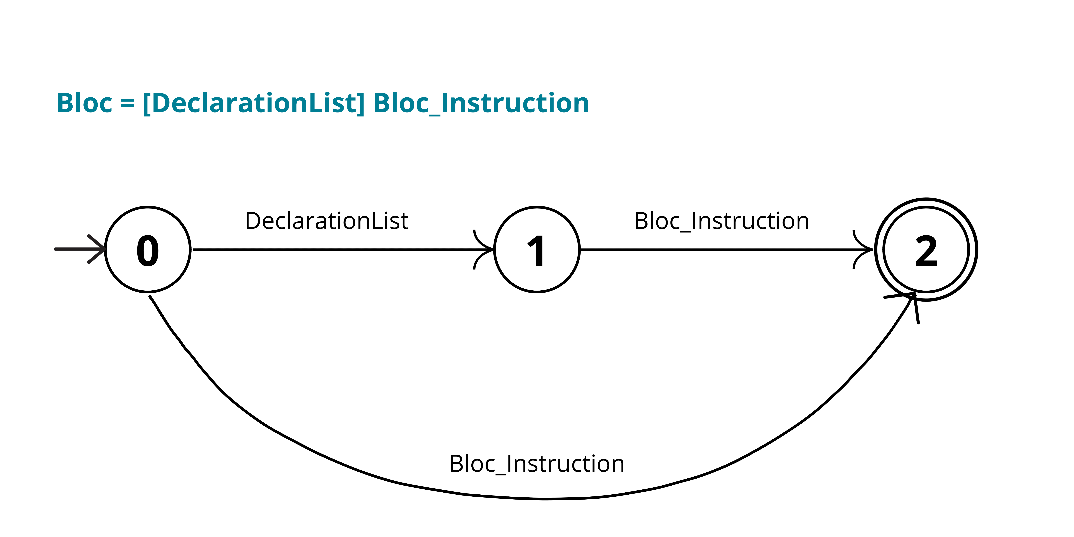
- Les règles sont écrites dans un format proche des productions **EBNF**, facilitant la compréhension.

**Les règles avec leurs automates associés**

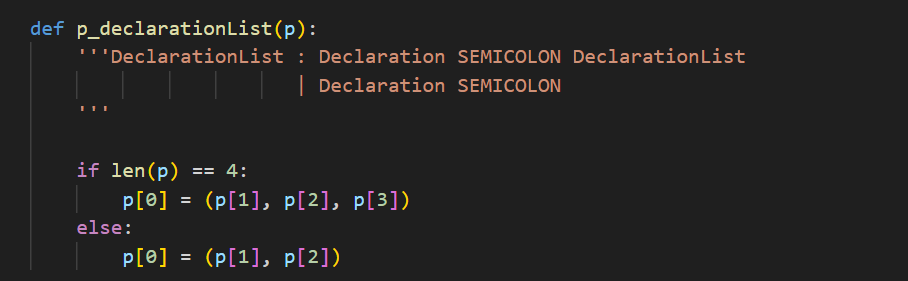
**Bloc = [ DeclarationList ] Bloc\_Instruction**

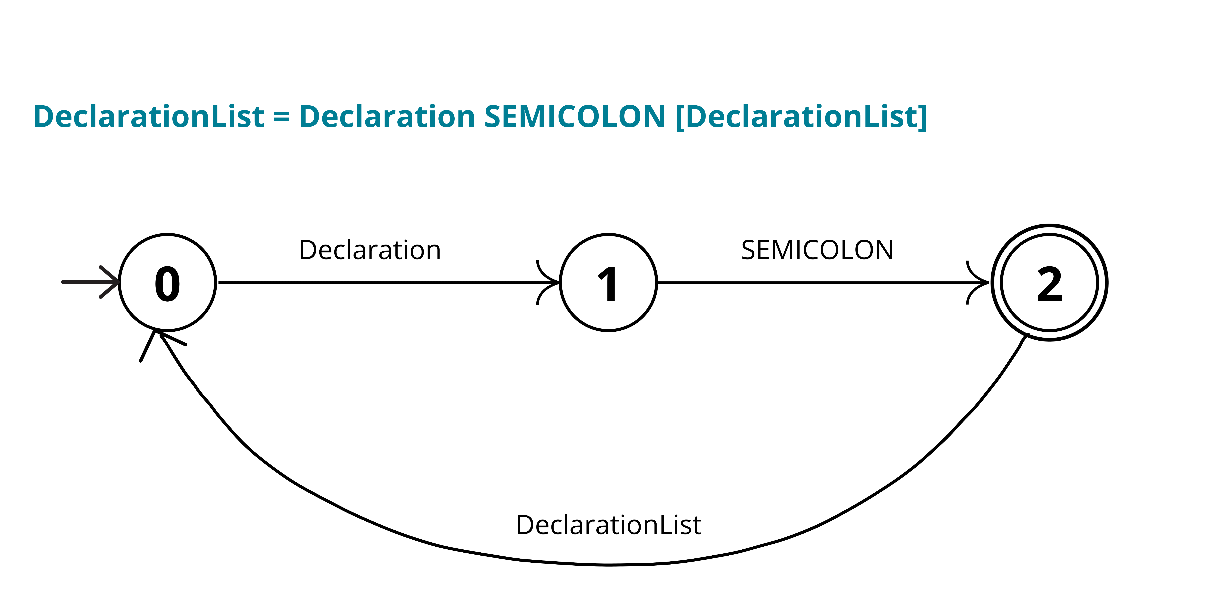


7

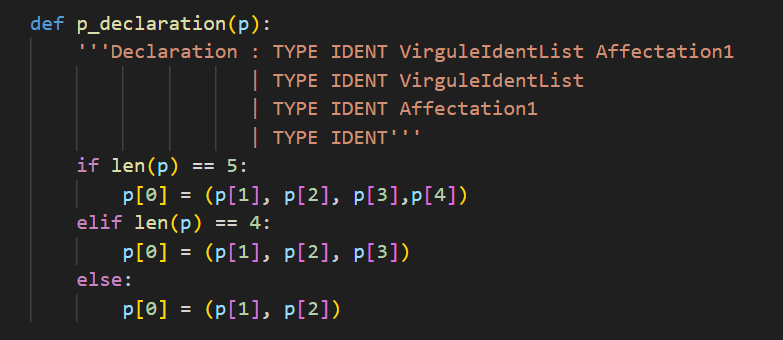


**DeclarationList = Declaration SEMICOLON {DeclarationList}**

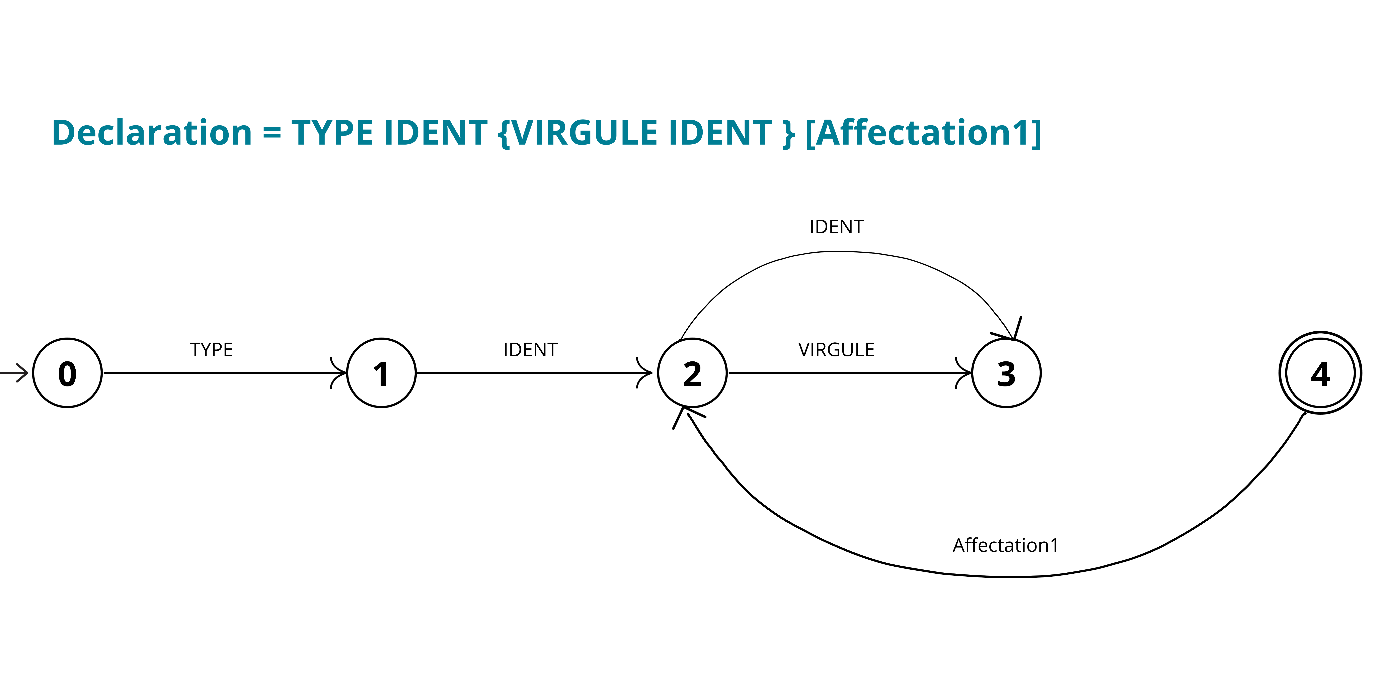




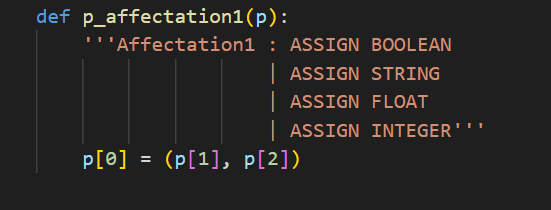
**Declaration = TYPE IDENT { VIRGULE IDENT } [ Affectation1 ]**

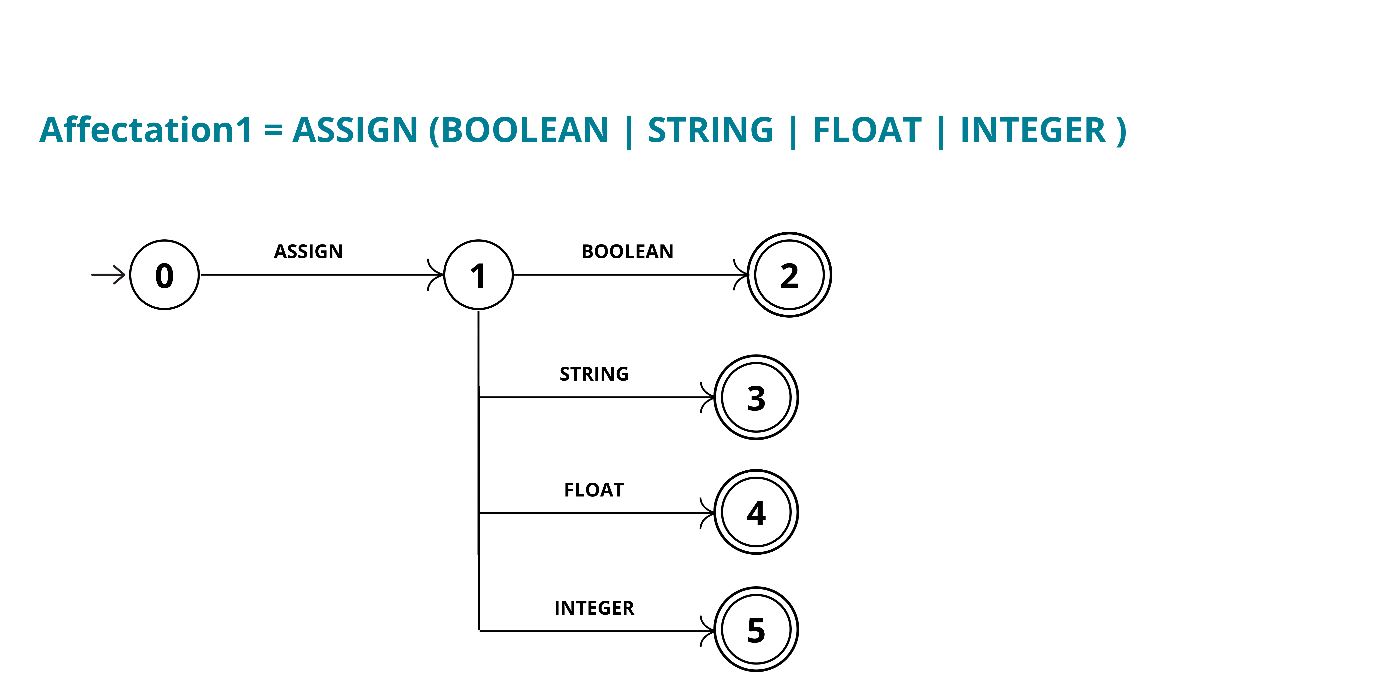
****

8

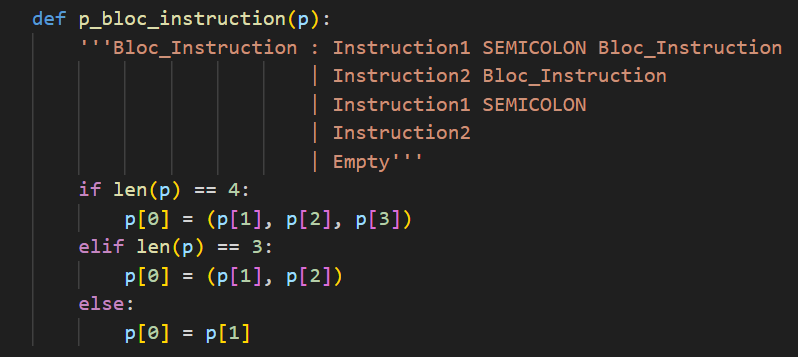
****

**Affectation1 = ASSIGN ( BOOLEAN | STRING | FLOAT | INTEGER )**

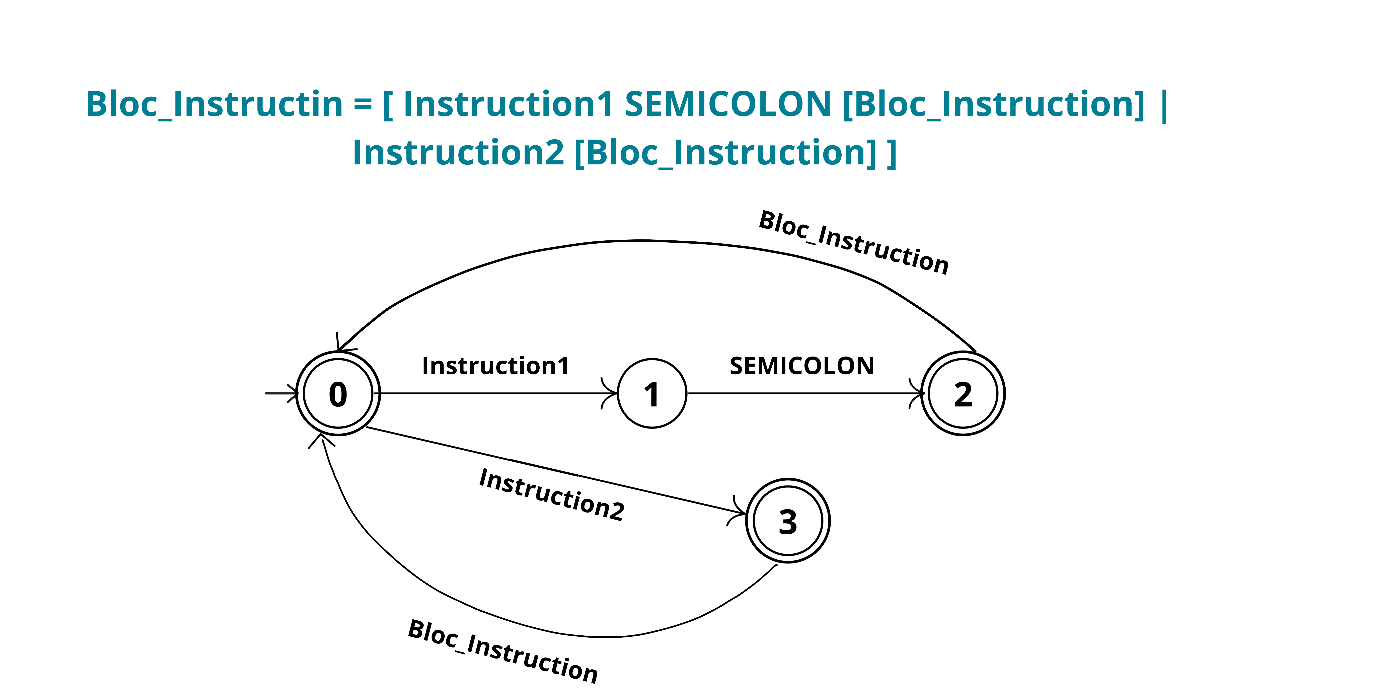
****

****

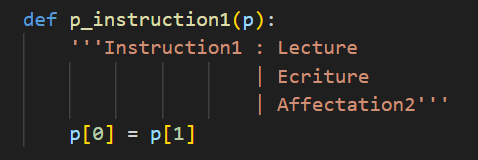
**Bloc\_Instruction = [ Instruction1 SEMICOLON [ Bloc\_Instruction ] | Instruction2 [ Bloc\_Instruction ] ]**

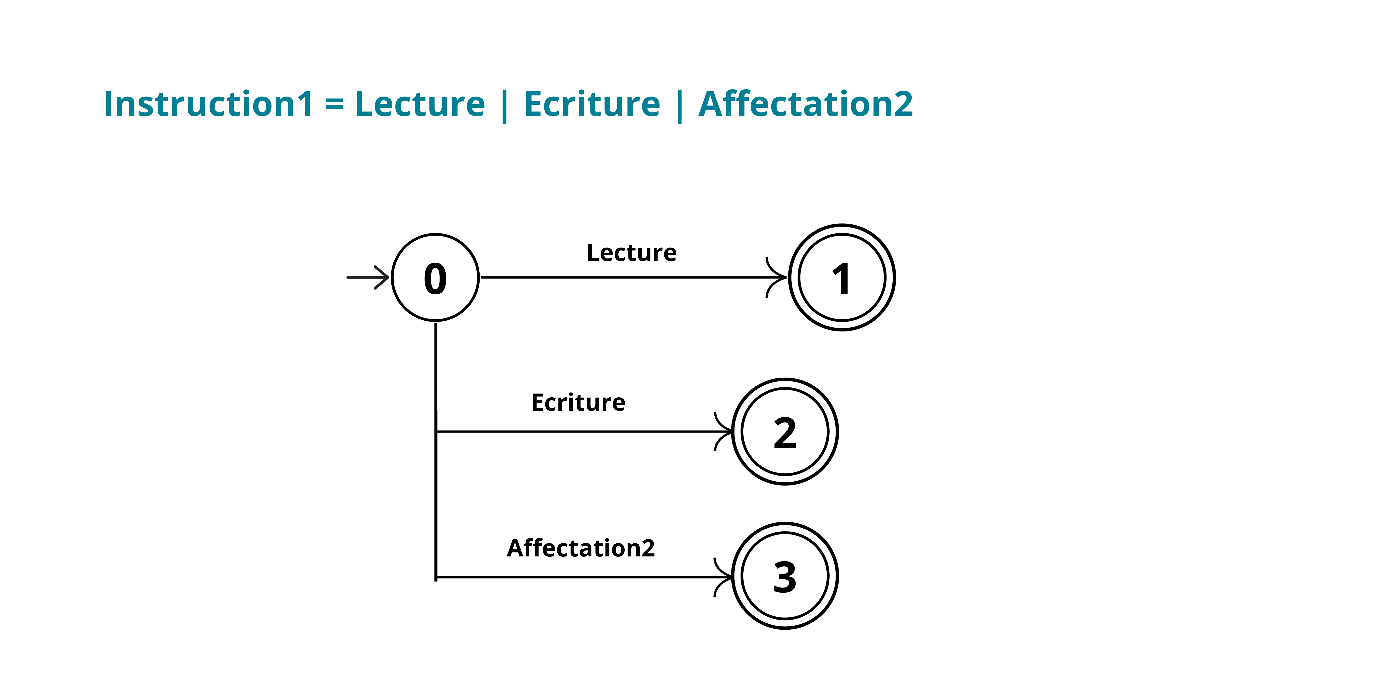
****

9

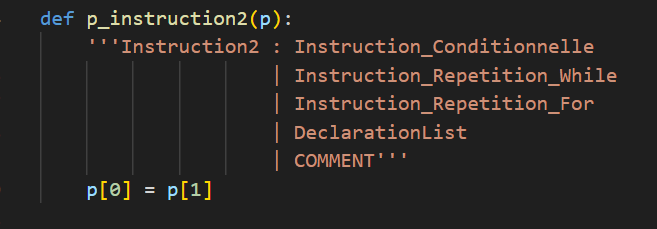
****

**Instruction1 = Lecture | Ecriture | Affectation2**

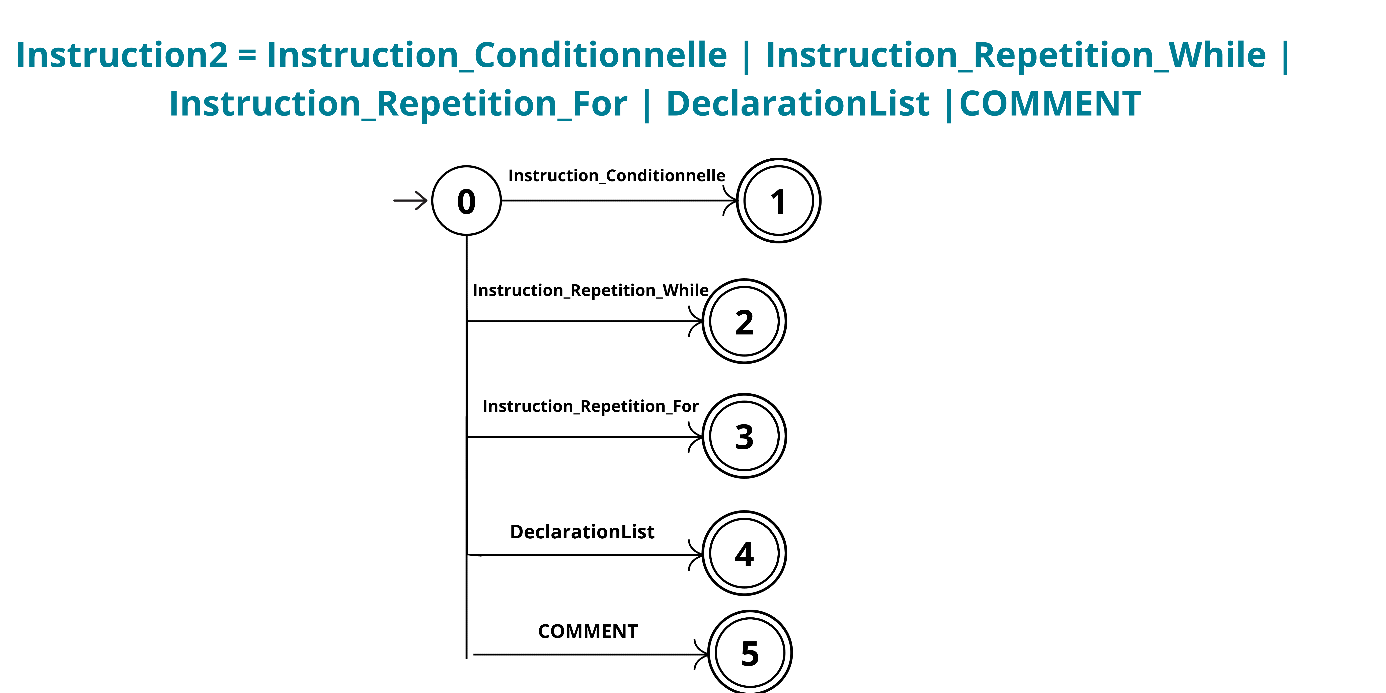
****

****

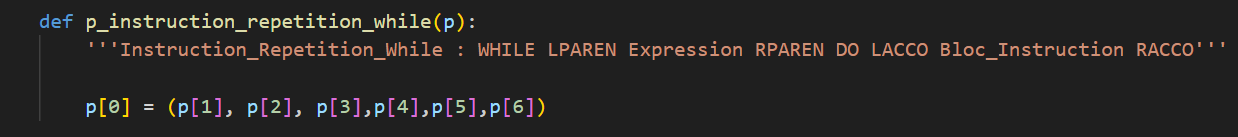
**Instruction2 = Instruction\_Conditionnelle | Instruction\_Repetition\_While | Instruction\_Repetition+For | DeclarationList | COMMENT**

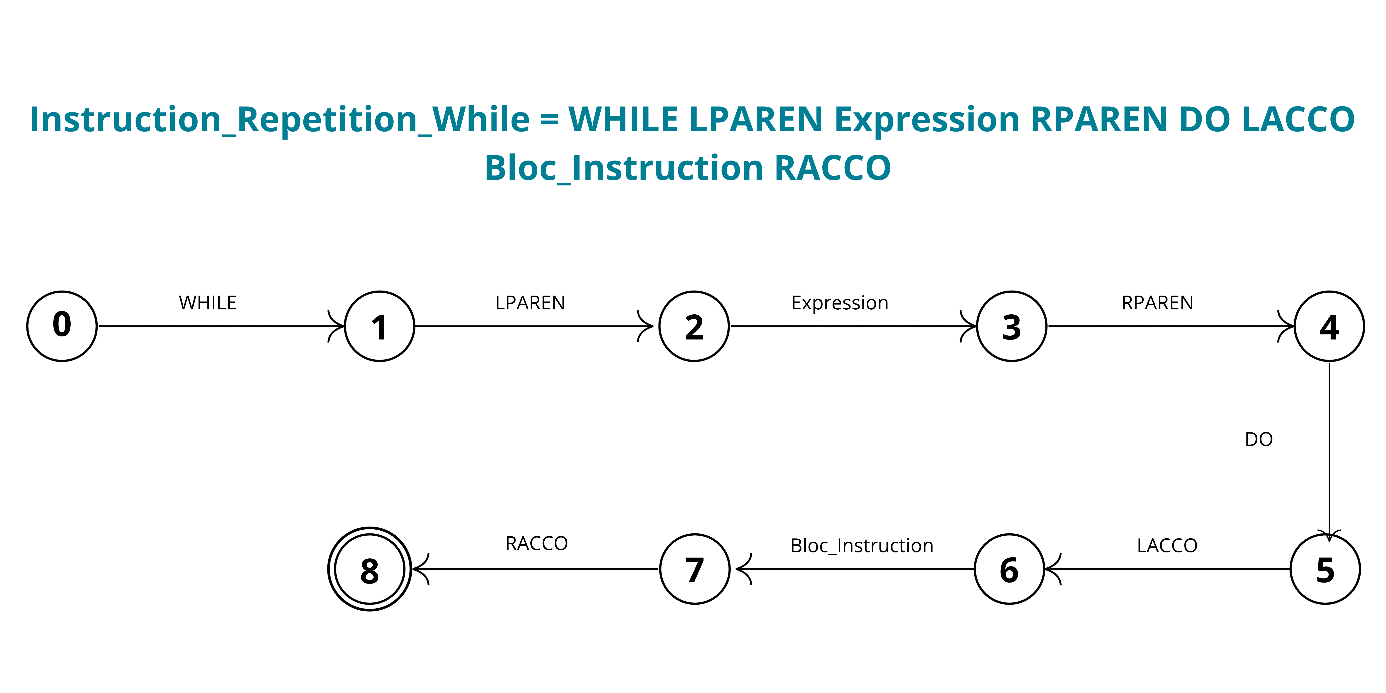
****

10

****

**Instruction\_Repetition\_While = WHILE LPAREN Expression RPAREN DO LACCO Bloc\_Instruction RACCO**

****

****

**Instruction\_Repetition\_For = FOR LPAREN [ TYPE ] IDENT Affectation1 SEMICOLON IDENT ( GT | LEQ | LT | GEQ | DIFFERENT | EQ ) Expression\_Simple SEMICOLON Affectation2 RPAREN (LACCO Bloc\_Instruction RACCO | Bloc\_Instruction )**

def p\_instruction\_repetition\_for(p):

    '''Instruction\_Repetition\_For : FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT LT Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT GT Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT LEQ Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT GEQ Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

11

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT DIFFERENT Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT EQ Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT LT Expression\_Simple SEMICOLON Affectation2 RPAREN  Bloc\_Instruction

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT GT Expression\_Simple SEMICOLON Affectation2 RPAREN  Bloc\_Instruction

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT LEQ Expression\_Simple SEMICOLON Affectation2 RPAREN Bloc\_Instruction

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT GEQ Expression\_Simple SEMICOLON Affectation2 RPAREN Bloc\_Instruction

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT DIFFERENT Expression\_Simple SEMICOLON Affectation2 RPAREN Bloc\_Instruction

                                  | FOR LPAREN TYPE IDENT Affectation1 SEMICOLON IDENT EQ Expression\_Simple SEMICOLON Affectation2 RPAREN Bloc\_Instruction

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT LT Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT GT Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT LEQ Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT GEQ Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT DIFFERENT Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT EQ Expression\_Simple SEMICOLON Affectation2 RPAREN LACCO Bloc\_Instruction RACCO

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT LT Expression\_Simple SEMICOLON Affectation2 RPAREN  Bloc\_Instruction

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT GT Expression\_Simple SEMICOLON Affectation2 RPAREN  Bloc\_Instruction

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT LEQ Expression\_Simple SEMICOLON Affectation2 RPAREN Bloc\_Instruction

12

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT GEQ Expression\_Simple SEMICOLON Affectation2 RPAREN Bloc\_Instruction

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT DIFFERENT Expression\_Simple SEMICOLON Affectation2 RPAREN Bloc\_Instruction

                                  | FOR LPAREN IDENT Affectation1 SEMICOLON IDENT EQ Expression\_Simple SEMICOLON Affectation2 RPAREN Bloc\_Instruction'''

    if len(p) == 15:

        p[0] = (p[1], p[2], p[3],p[4],p[5],p[6],p[7],p[8],p[9],p[10],p[11],p[12],p[13],p[14])

    elif len(p) == 14:

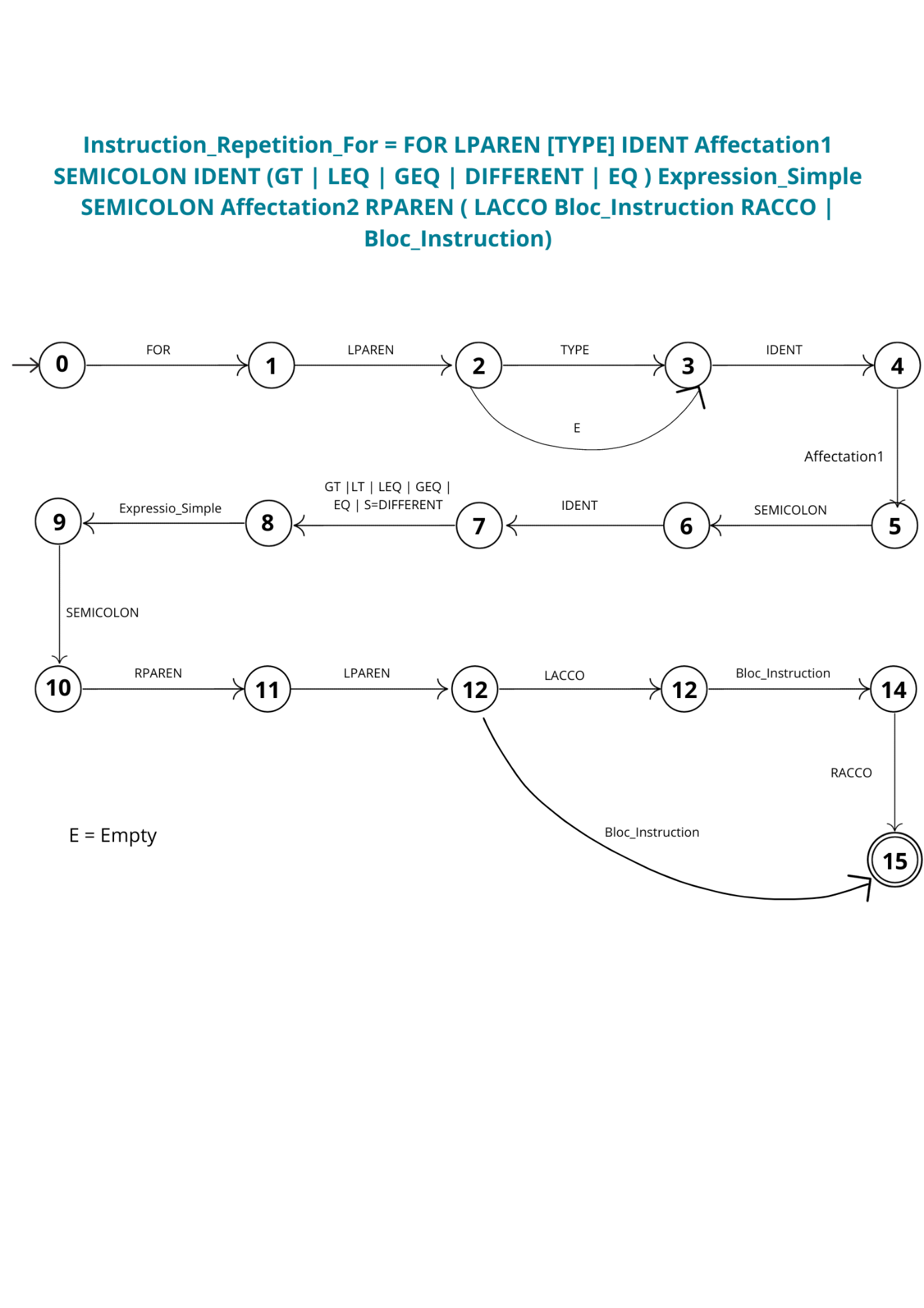
        p[0] = (p[1], p[2], p[3],p[4],p[5],p[6],p[7],p[8],p[9],p[10],p[11],p[12],p[13])

    elif len(p) == 13:

        p[0] = (p[1], p[2], p[3],p[4],p[5],p[6],p[7],p[8],p[9],p[10],p[11],p[12])

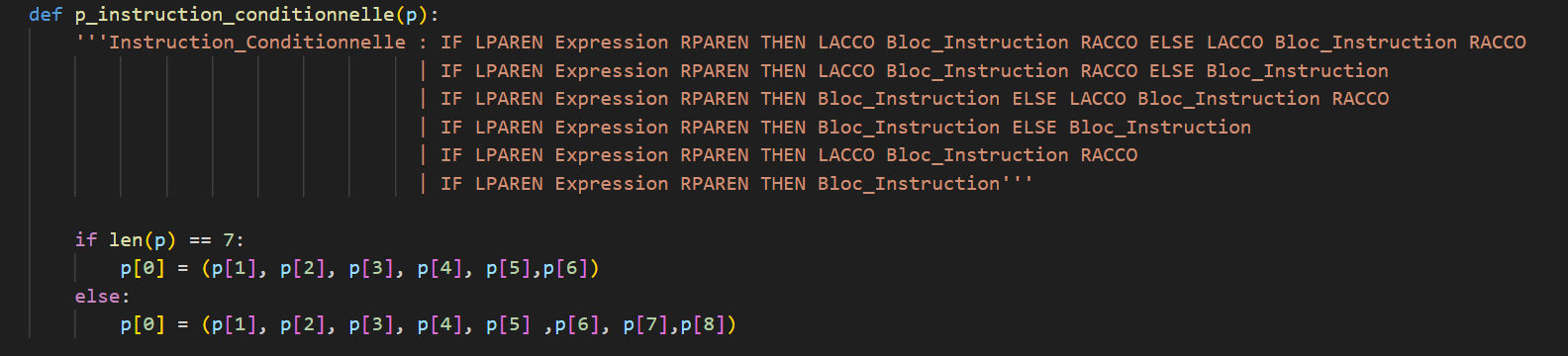
    else:

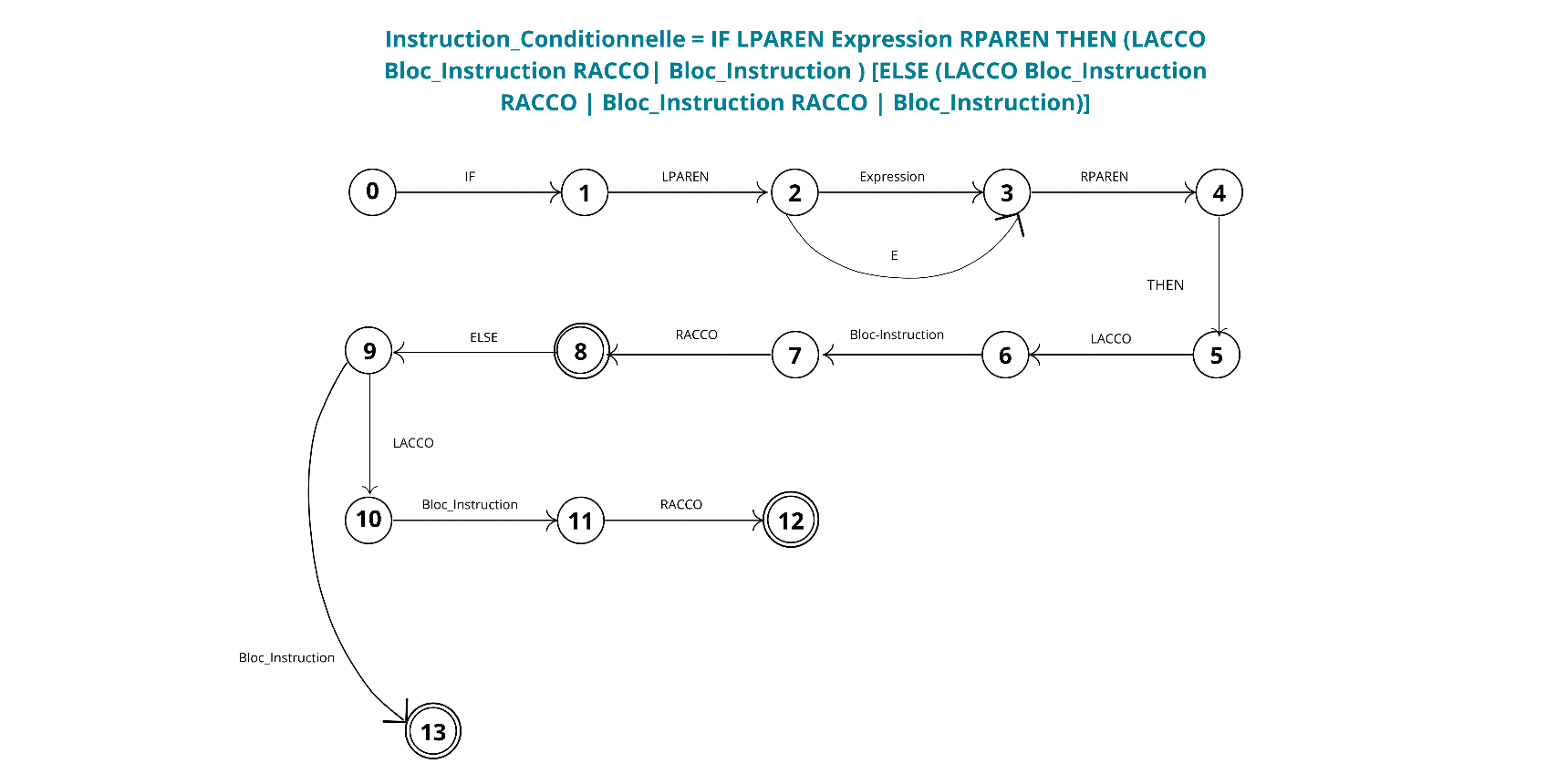
        p[0] = (p[1], p[2], p[3],p[4],p[5],p[6],p[7],p[8],p[9],p[10],p[11])

****

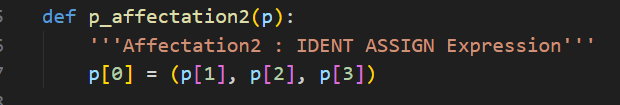
**Instruction\_Conditionnelle = IF LPAREN Expression RPAREN THEN (LACCO Bloc\_Instruction RACCO | Bloc\_Instruction) ]**

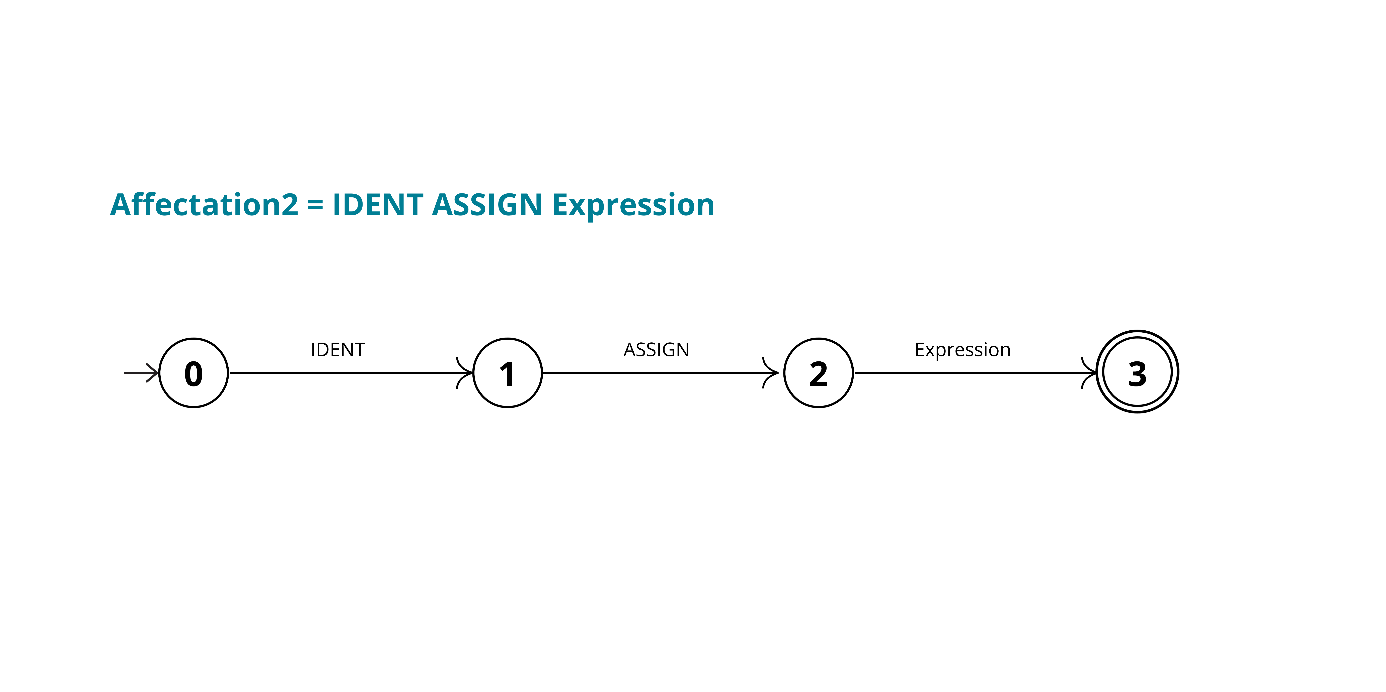
13

****

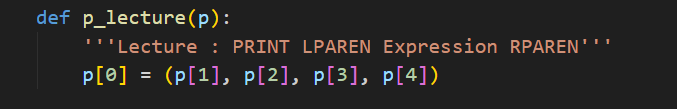
****

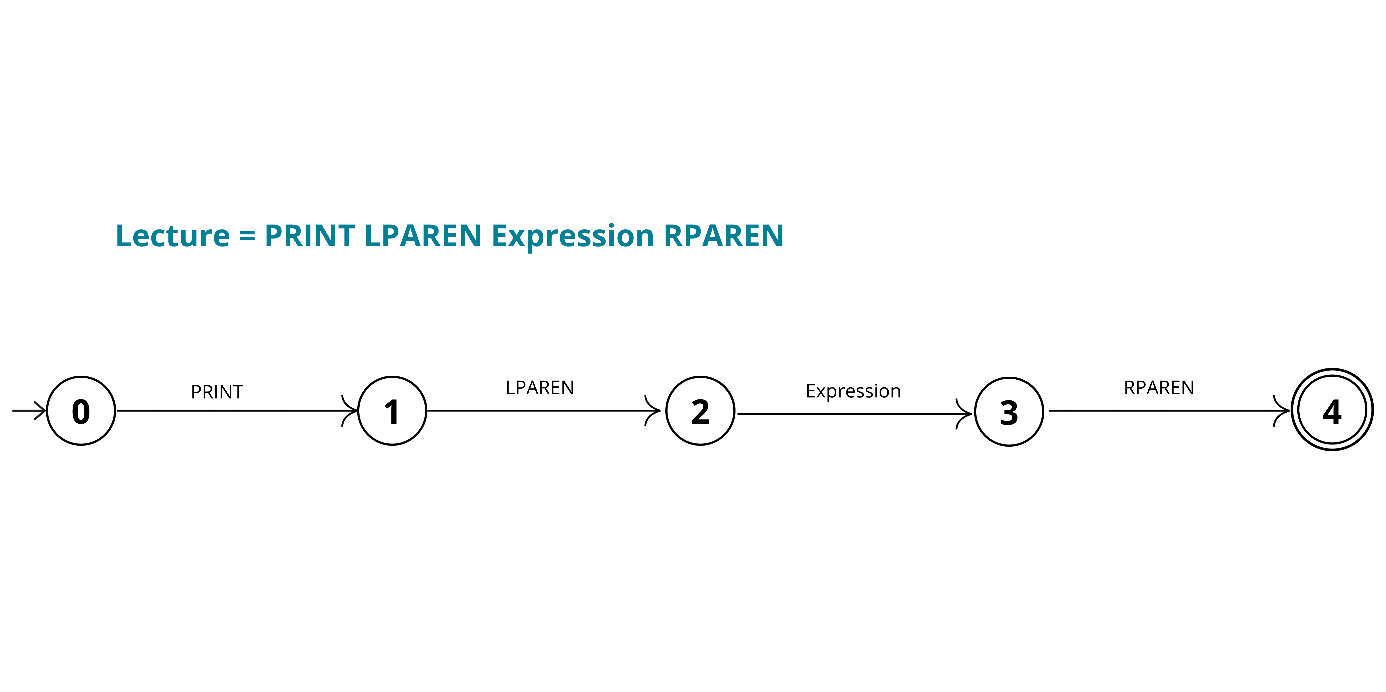
**Affectation2 = IDENT ASSIGN Expression**

****

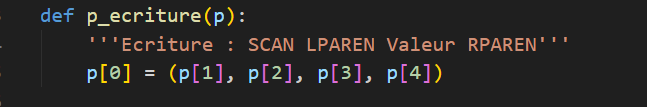
****

**Lecture = PRINT LPAREN Expression RPAREN**

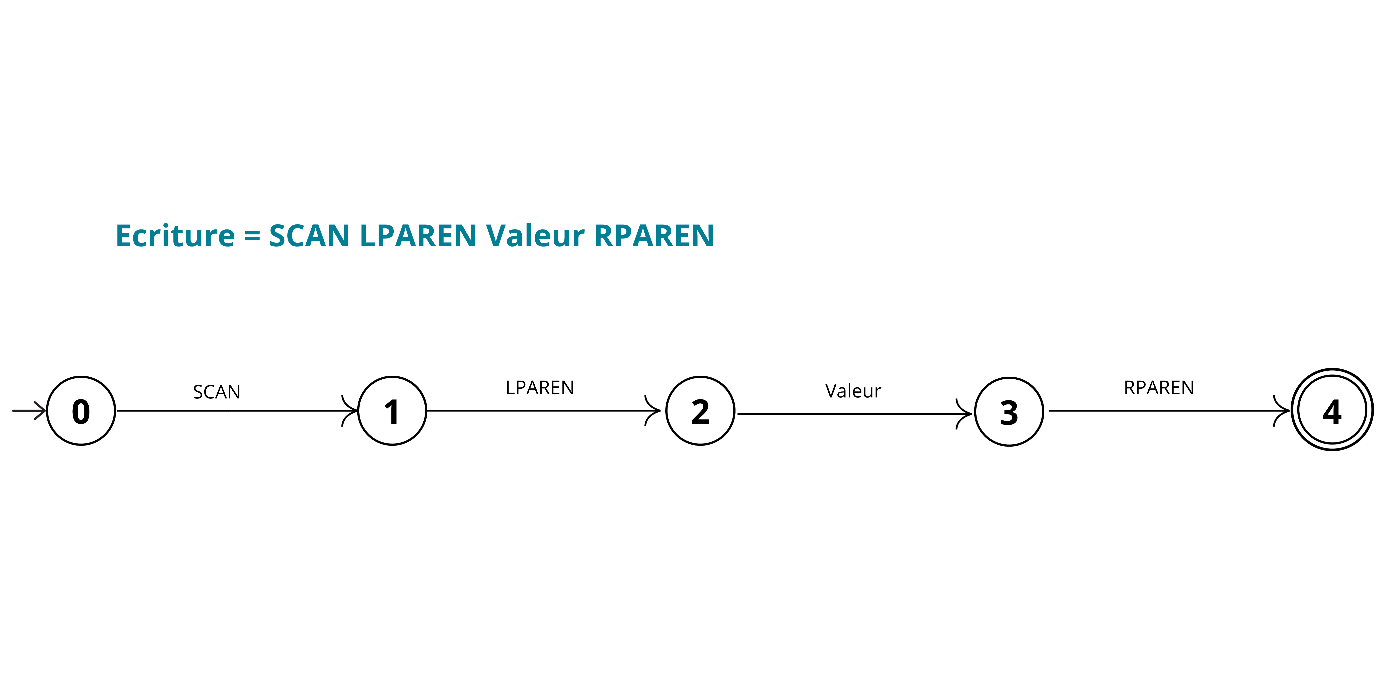
****

****

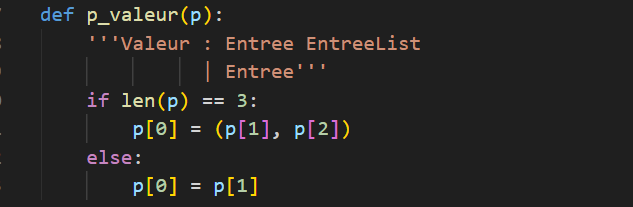
**Ecriture = SCAN LPAREN Valeur RPAREN**

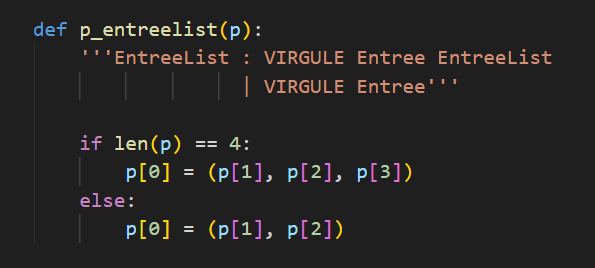
****

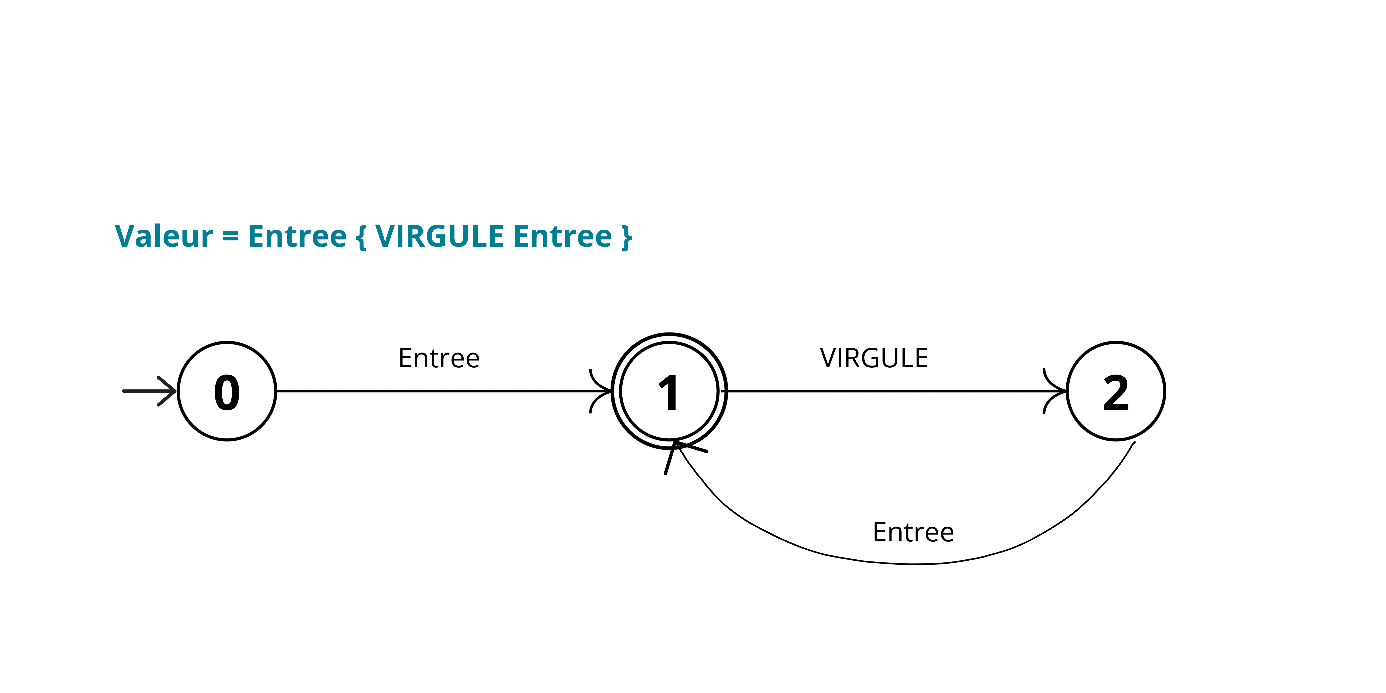
14

****

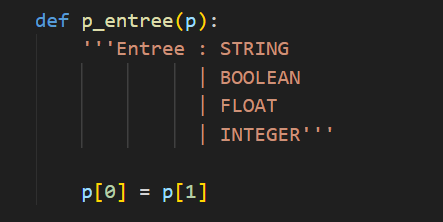
**Valeur = Entree { VIRGULE Entree }**

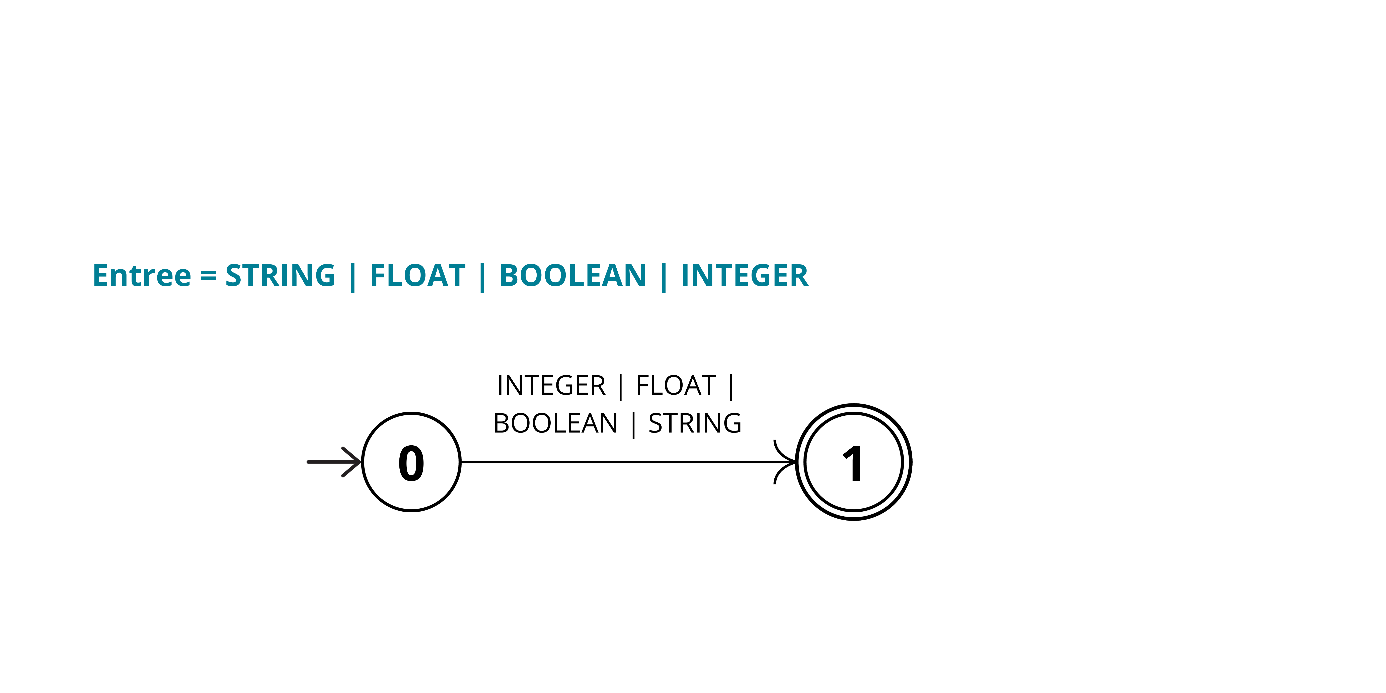
****

****

****

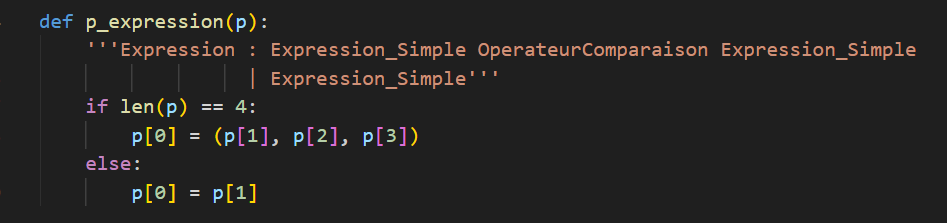
**Entree = STRING | BOOLEAN | FLOAT | INTEGER**

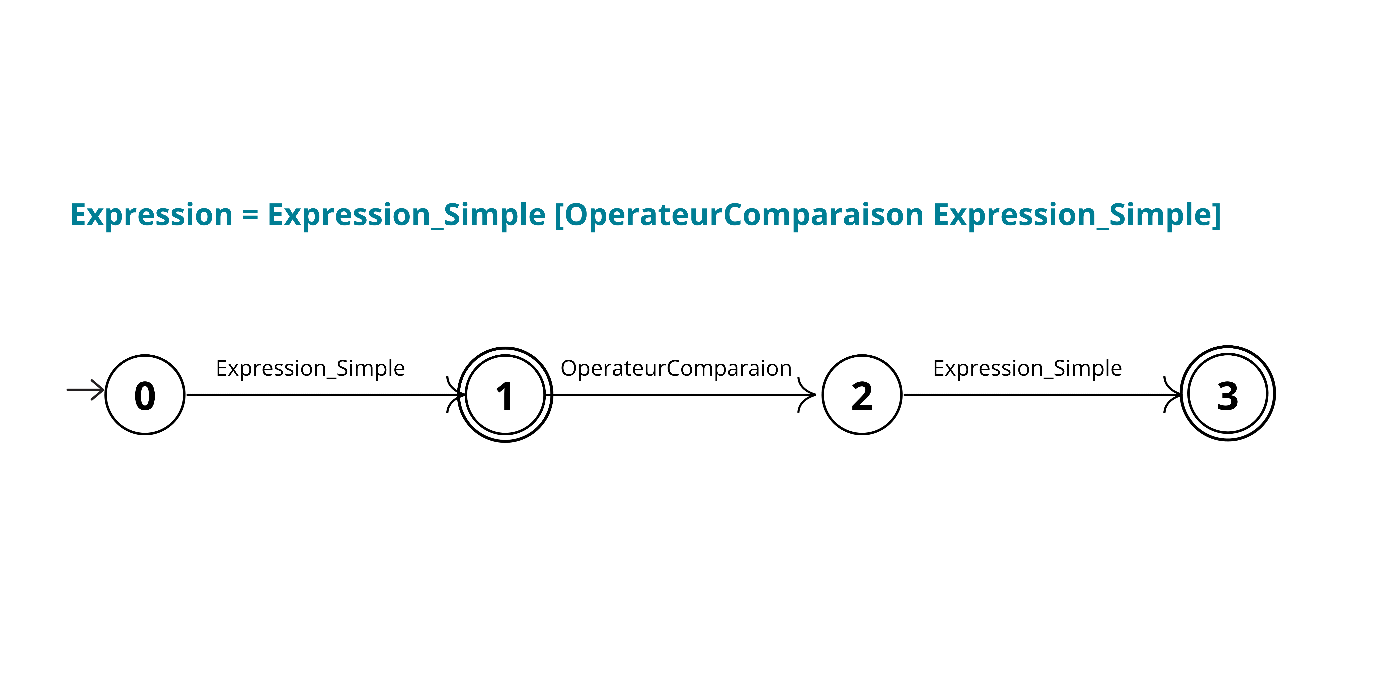
****

****

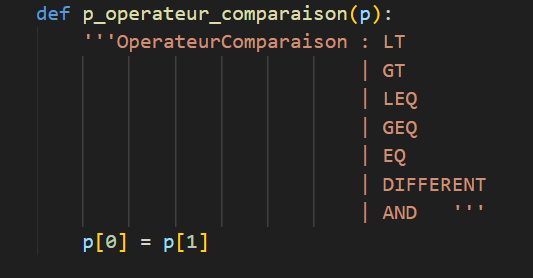
**Expression = Expression\_Simple [ OperateurComparaison Expression\_Simple ]**

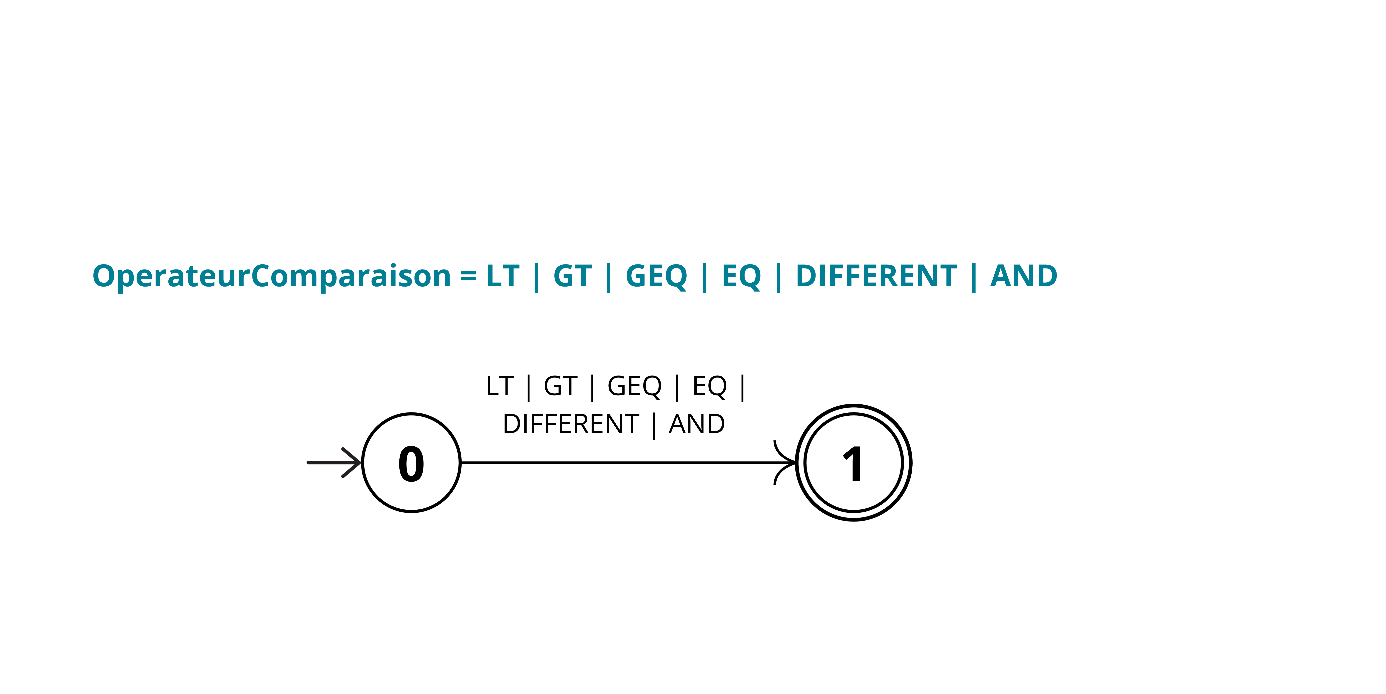
15

****

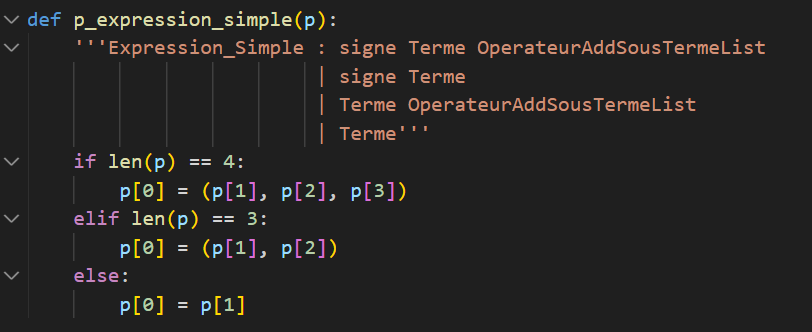
****

**OperateurComparaison = LT | GT | GEQ | EQ | DIFFERENT | AND**

****

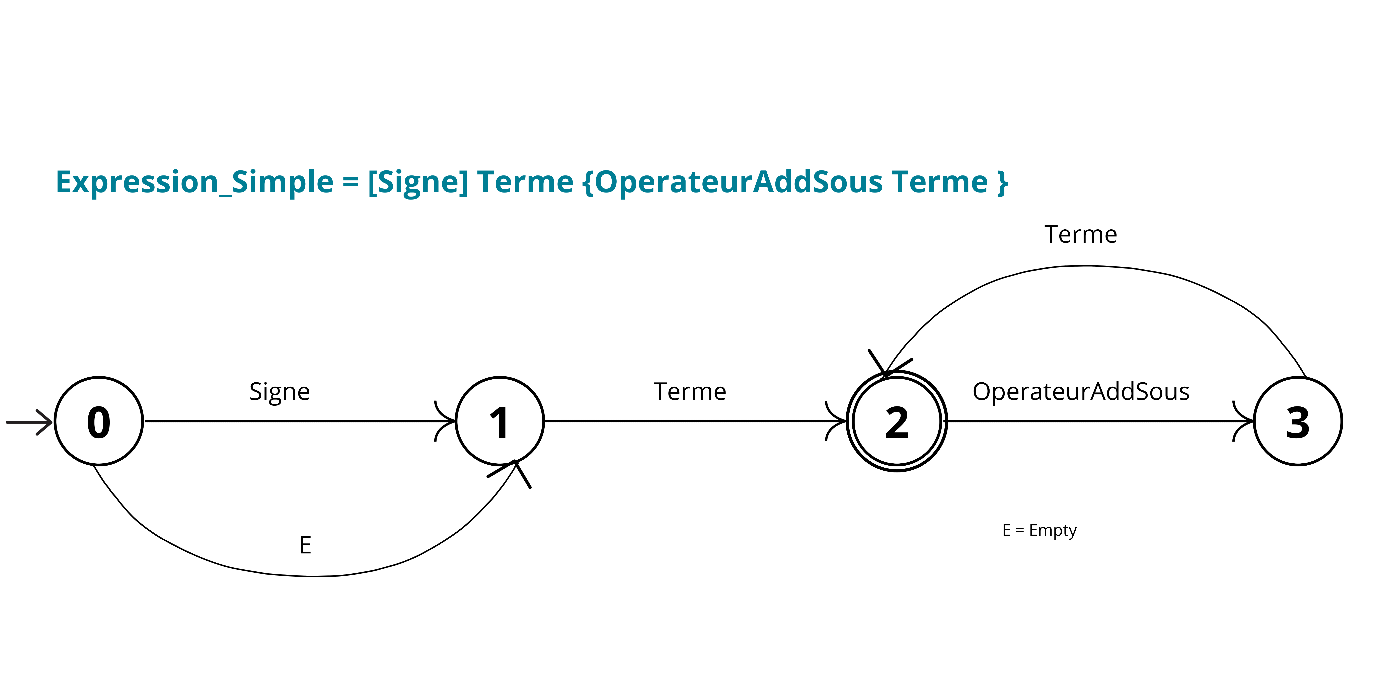
****

**Expression\_Simple = [ Signe ] Terme { OperateurAddSous Terme }**

****

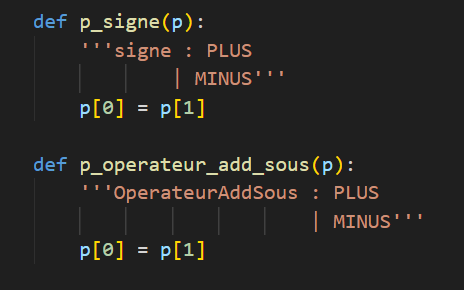
16

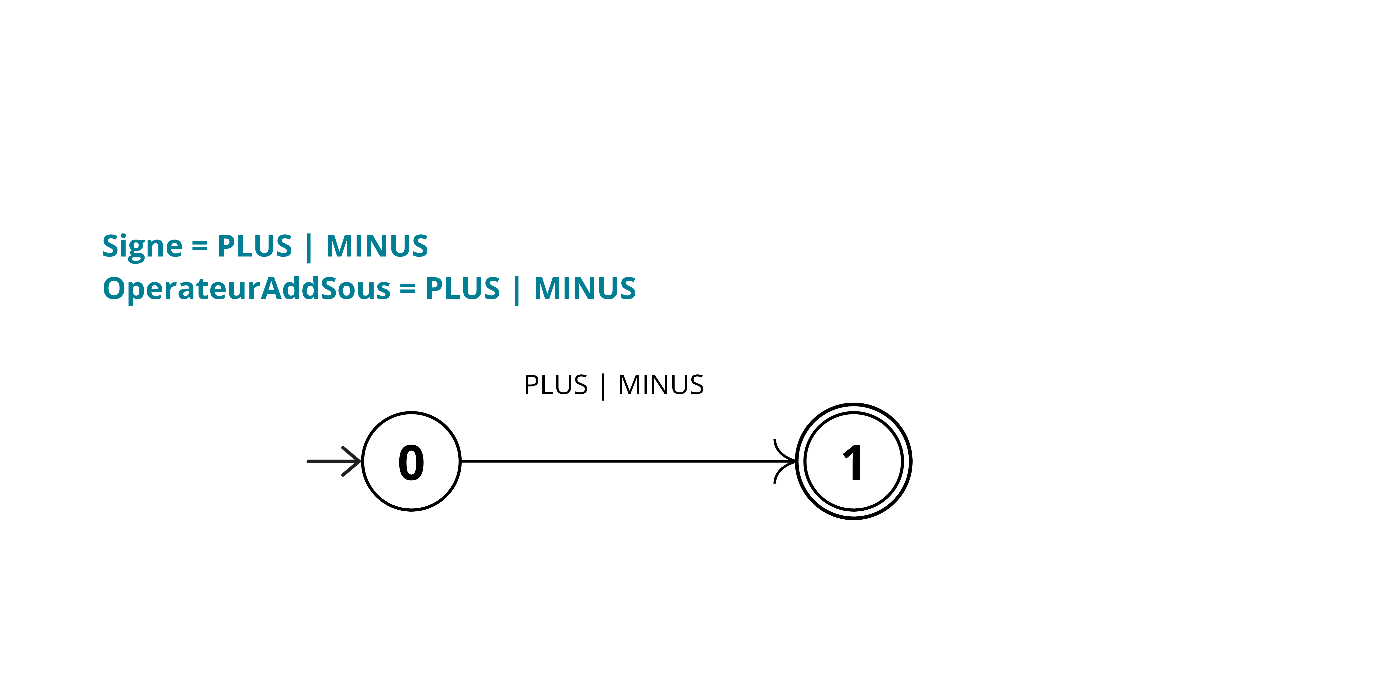
**XXXXXXXX**

****

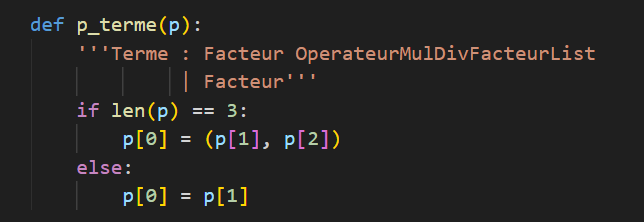
**Signe = PLUS | MINUS**

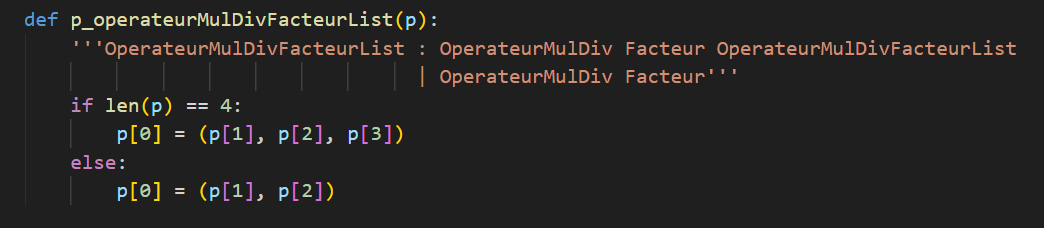
**OperateurAddSous = PLUS | MINUS**

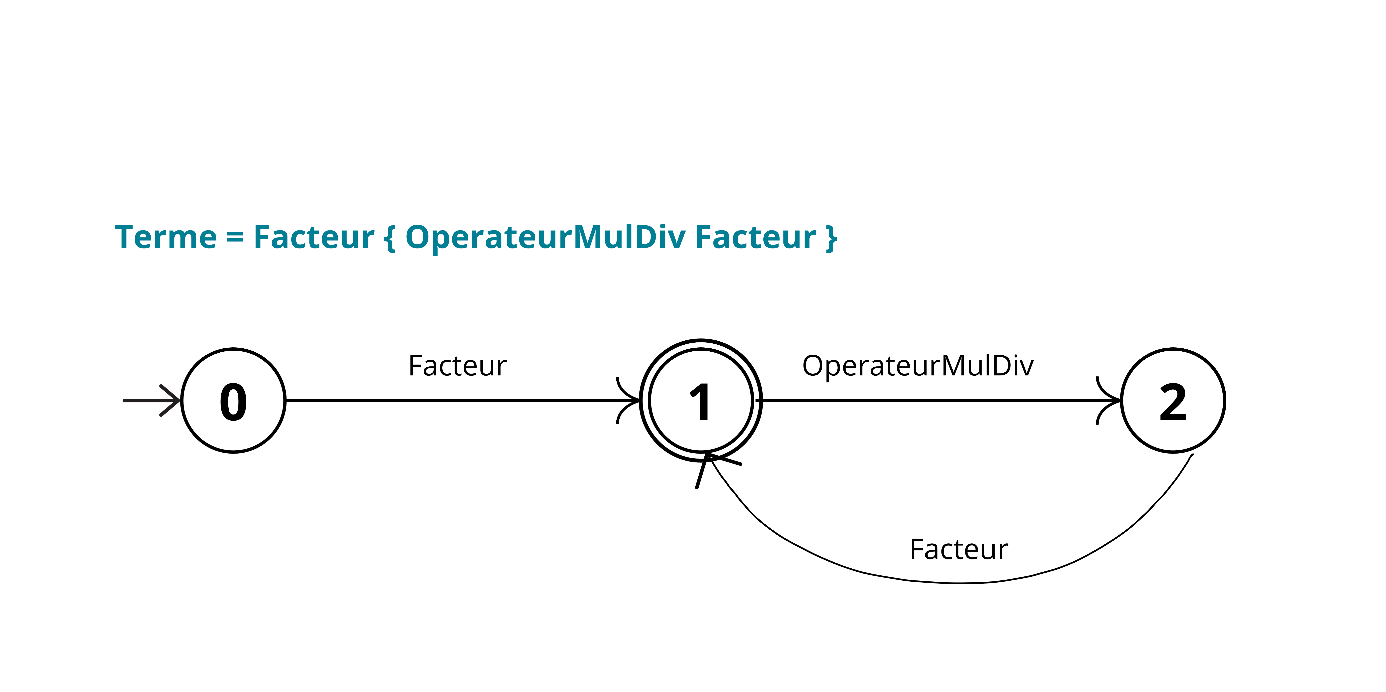
****

****

**Terme = Facteur { OperateurMulDiv Facteur }**

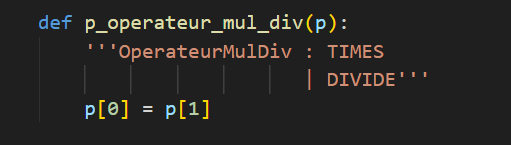
****

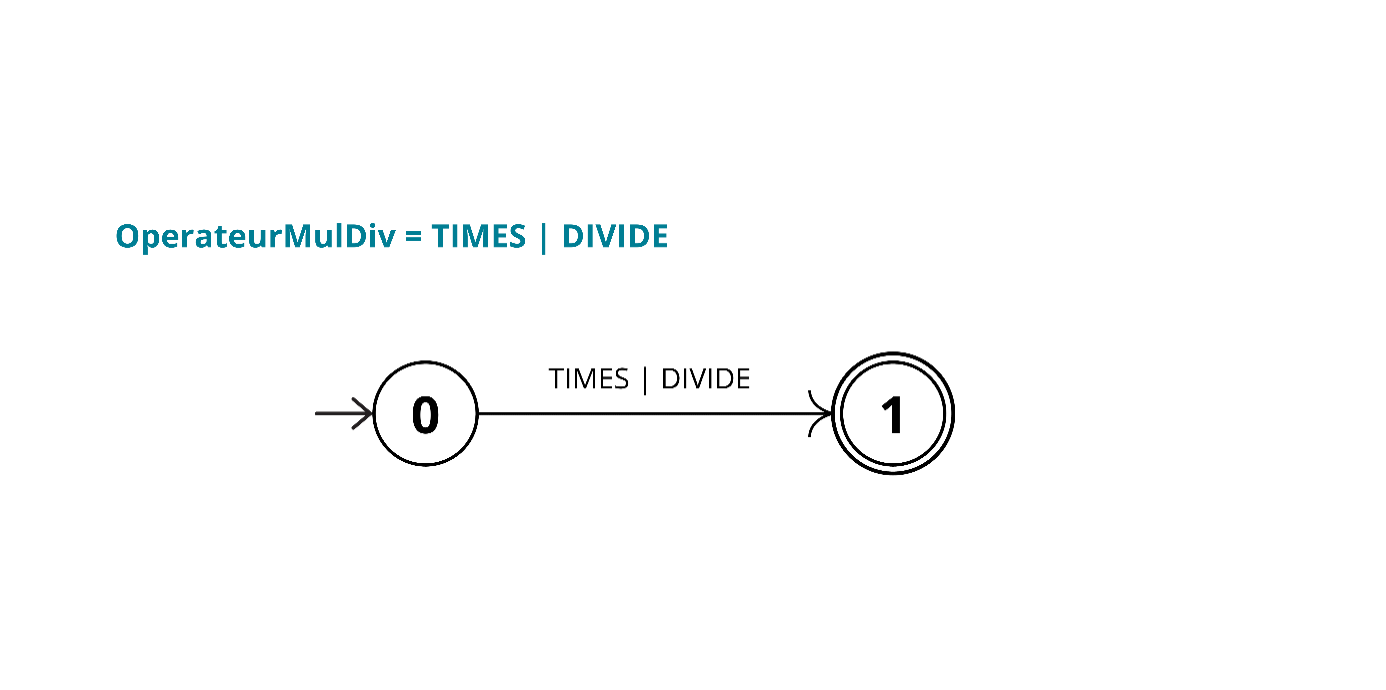
****

****

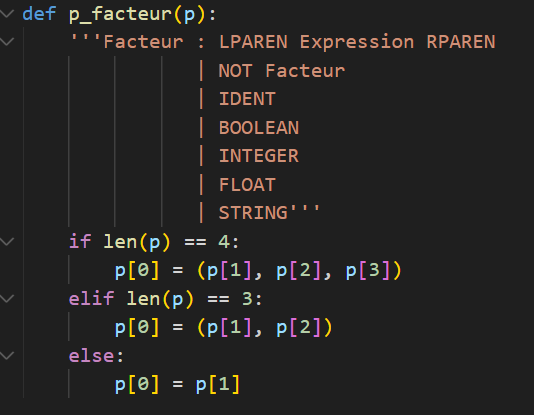
17

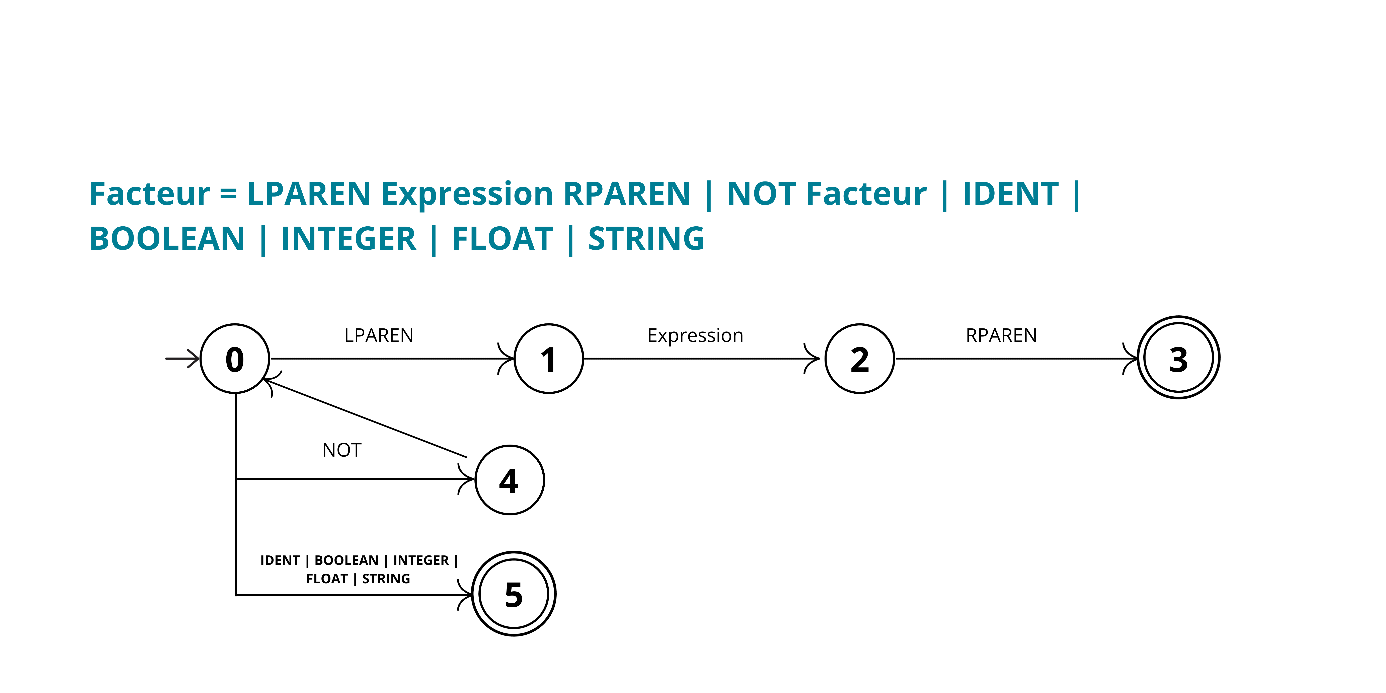
**OperateurMulDiv = TIMES | DIVIDE**

****

****

**Facteur = LPAREN Expression RPAREN | NOT Facteur | IDENT | BOOLEAN | INTEGER | STRING | FLOAT**

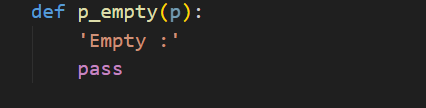
****

****

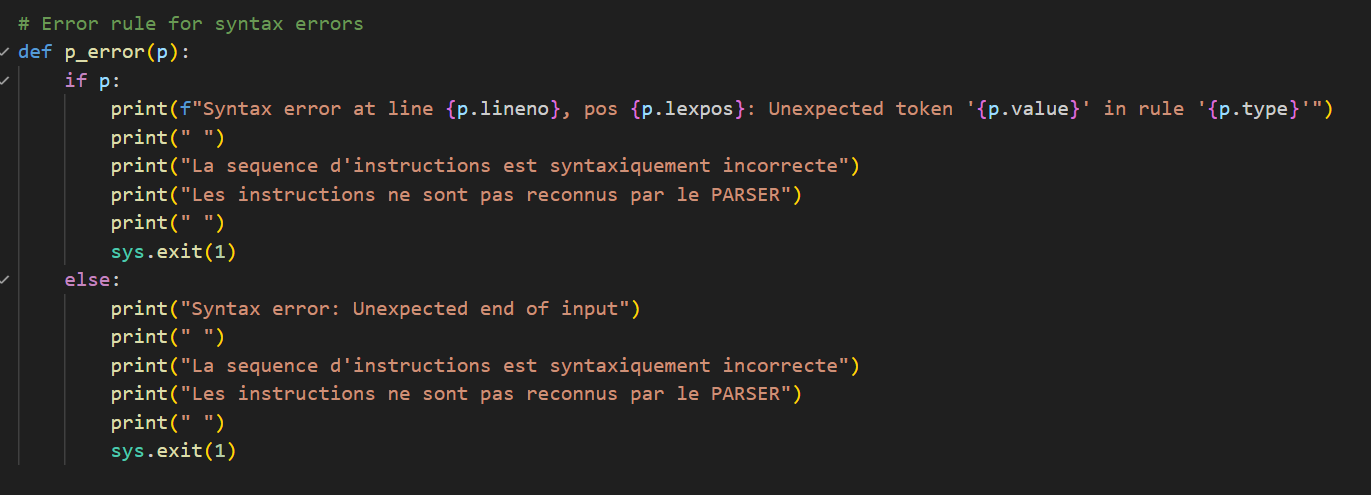
18

**Empty = E**

`p\_empty` est une règle de la grammaire qui représente une séquence vide. Lorsqu'elle est utilisée, elle ne génère rien (`pass`). En d'autres termes, elle indique qu'une partie de la syntaxe peut être absente.

****

**Gestion des erreurs syntaxiques :** La fonction `p\_error` est une règle spéciale appelée en cas d'erreur syntaxique. Elle affiche un message détaillé sur l'erreur (ligne, position, type de token) et termine le programme avec `sys.exit(1)`.

****

**Analyse lexicale avec le Lexer :**

Cette partie utilise l'analyseur lexical (`lexer`) pour découper le code source (`data`) en tokens. Chaque token est ensuite imprimé avec des informations sur sa ligne, sa position et sa valeur.

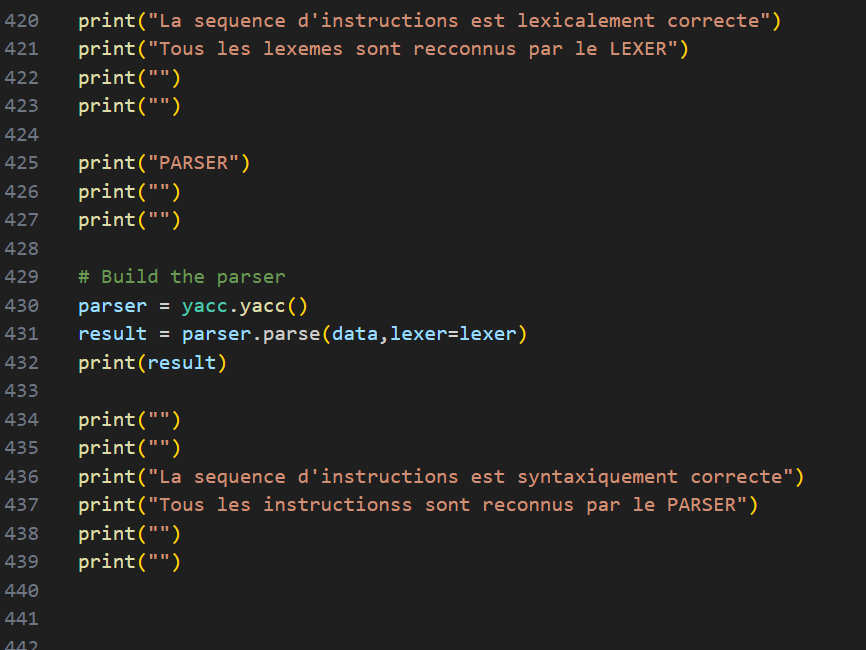
19

****

**Analyse syntaxique avec le Parser :**

Cette partie construit l'analyseur syntaxique (`parser`) et l'utilise pour analyser l’entée de l’utilisateur qui est sous forme d’une séquence d’instructions (`data`). Le résultat de l'analyse est imprimé. Si l'analyse syntaxique est réussie, `result` contient une représentation arborescente de la structure de la séquence d’instructions ‘data’ représentant le processus shift/reduce que l’analyseur syntaxique(parser) avec laquelle il s’est assuré que ‘data’ est conforme à la grammaire.

20

****

**Conclusion :**

En résumé, ce code combine un **analyseur lexical (lexer)** et un **analyseur syntaxique (parser)** pour traiter un code source en vérifiant sa validité syntaxique tout en respectant la **grammaire EBNF**. En cas d'erreur, des messages détaillés sont affichés, indiquant la nature et l'emplacement des erreurs.

21