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, ...

FinalProject.py > ...

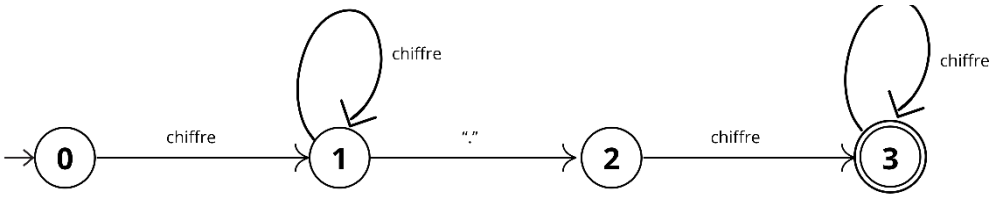
```python
import sys
import ply.lex as lex
import ply.yacc as yacc

tokens = [
    'IF',
    'TYPE',
    'ASSIGN',
    'BOOLEAN',
    'STRING',
    'FLOAT',
    'INTEGER',
    'THEN',
    'ELSE',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LT',
    'GT',
    'LEQ',
    'GEQ',
    'EQ',
    'LPAREN',
    'RPAREN',
    'LACCO',
    'RACCO',
    'NOT',
    'SEMICOLON',
    'COMMENT',
    'VIRGULE',
    'DIFFERENT',
    'AND',
    'PRINT',
    'SCAN',
    'WHILE',
    'DO',
    'FOR',
    'IDENT'
]

# Regular expression rules for tokens
t_IF = r'if'
t_TYPE = r'\b(?:bool|string|float|int)\b'
t_ASSIGN = r'='
t_BOOLEAN = r'\b(?:true|false)\b'
t_STRING = r'\"([^\\\n]|(\\.))*?\"'
t_FLOAT = r'\d+\.\d+'
t_INTEGER = r'\d+'
t_THEN = r'then'
t_ELSE = r'else'
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LT = r'<'
```

FinalProject.py > ...

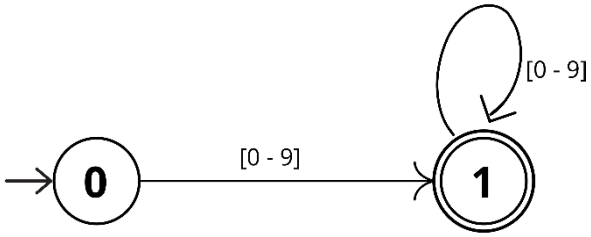
```
46    t_BOOLEAN = r'\b(?:true|false)\b'
47    t_STRING = r'\"([^\\\n]|(\\.))*?\"'
48    t_FLOAT = r'\d+\.\d+'
49    t_INTEGER = r'\d+'
50    t_THEN = r'then'
51    t_ELSE = r'else'
52    t_PLUS = r'\+'
53    t_MINUS = r'-'
54    t_TIMES = r'\*'
55    t_DIVIDE = r'/'
56    t_LT = r'<'
57    t_GT = r'>'
58    t_LEQ = r'<='
59    t_GEQ = r'>='
60    t_EQ = r'=='
61    t_LACCO = r'{'
62    t_RACCO = r'}'
63    t_DIFFERENT = r'!='
64    t_AND = r'&&'
65    t_LPAREN = r'\('
66    t_RPAREN = r'\)'
67    t_NOT = r'!'
68    t_SEMICOLON = r';'
69    t_VIRGULE = r','
70    t_COMMENT = r'\/\/.*'
71    t_PRINT = r'print'
72    t_SCAN = r'scan'
73    t_WHILE = r'while'
74    t_DO = r'do'
75    t_FOR=r'for'
76    t_IDENT = r'(?!if|else|while|do|for|then|int|float|string|print|scan)[a-zA-Z_][a-zA-Z0-9_]*'
77
78
```

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}



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

```python
t_ignore = ' \t'

# Track line numbers and positions
lexer_line_start = 1
lexer_pos_in_line = 1

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
    global lexer_line_start, lexer_pos_in_line
    lexer_line_start = t.lexer.lineno
    lexer_pos_in_line = 1

def t_error(t):
    global lexer_pos_in_line
    print(" ")
    print(f"erreur lexicale: lexeme non reconnu par l'analyseur lexical '{t.value[0]}' at line {t.lineno}, pos {lexer
    print(" ")
    t.lexer.skip(1)
    sys.exit(1)



# Define the grammar rules with actions
```

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.

2. Analyseur syntaxique (Parser) :

Règles de grammaire :

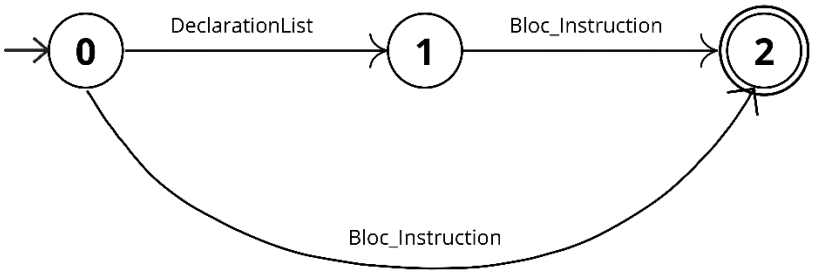
- Définition **des règles de grammaire** pour différentes constructions syntaxiques, telles que les déclarations, les instructions conditionnelles, les boucles, etc.

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

Les règles avec leurs automates associés

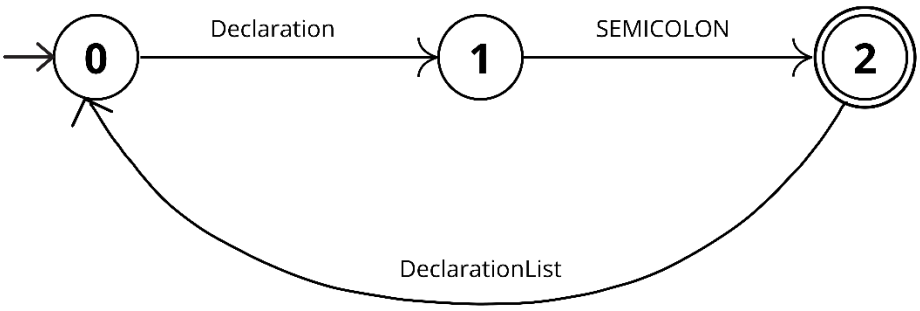
Bloc = [DeclarationList] Bloc_Instruction

```python
# Define the grammar rules with actions

def p_bloc(p):
    '''Bloc : Bloc_Instruction
            | DeclarationList Bloc_Instruction'''
    if len(p) == 3:
        p[0] = (p[1], p[2])
    else:
        p[0] = p[1]
```



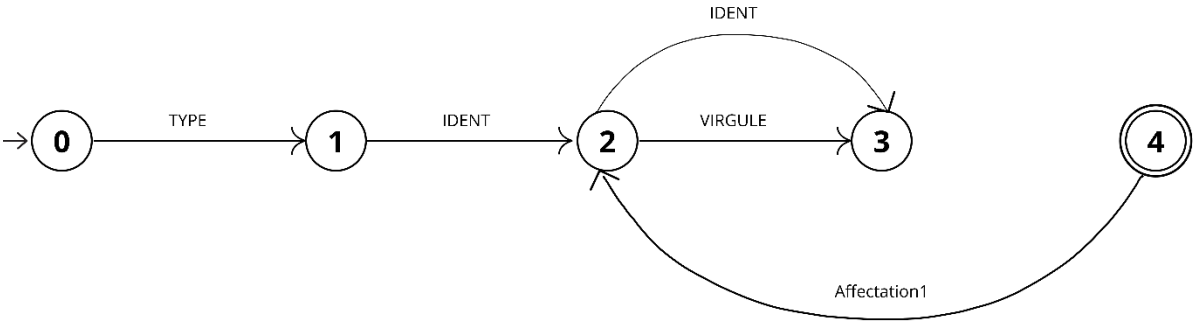
DeclarationList = Declaration SEMICOLON {DeclarationList}

```
def p_declarationList(p):
    '''DeclarationList : Declaration SEMICOLON DeclarationList
                       | Declaration SEMICOLON
    '''

    if len(p) == 4:
        p[0] = (p[1], p[2], p[3])
    else:
        p[0] = (p[1], p[2])
```



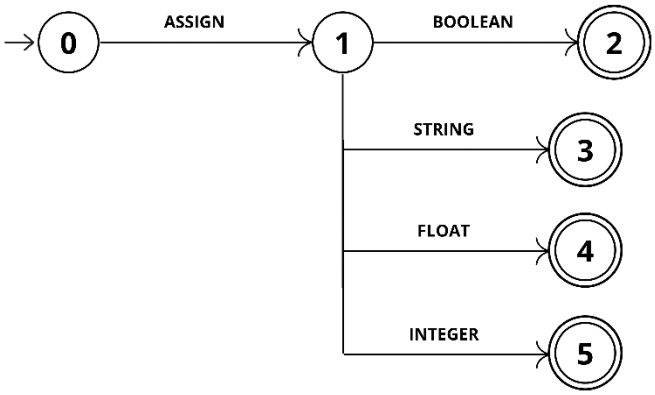
Declaration = TYPE IDENT { VIRGULE IDENT } [Affectation1]

```
def p_declaration(p):
    '''Declaration : TYPE IDENT VirguleIdentList Affectation1
                   | TYPE IDENT VirguleIdentList
                   | TYPE IDENT Affectation1
                   | TYPE IDENT'''
    if len(p) == 5:
        p[0] = (p[1], p[2], p[3],p[4])
    elif len(p) == 4:
        p[0] = (p[1], p[2], p[3])
    else:
        p[0] = (p[1], p[2])
```



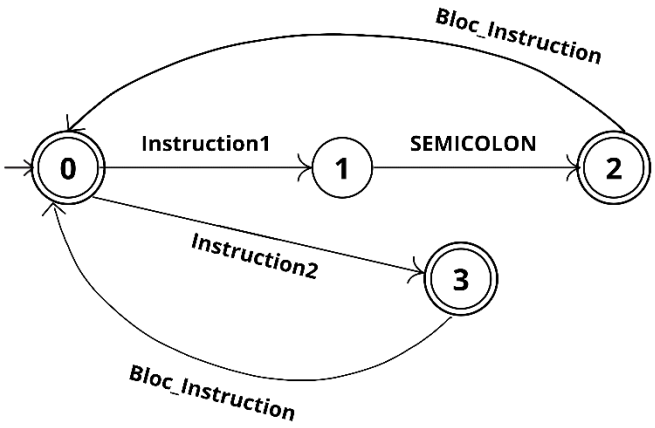
Affectation1 = ASSIGN (BOOLEAN | STRING | FLOAT | INTEGER)

```
def p_affectation1(p):
    '''Affectation1 : ASSIGN BOOLEAN
                    | ASSIGN STRING
                    | ASSIGN FLOAT
                    | ASSIGN INTEGER'''
    p[0] = (p[1], p[2])
```



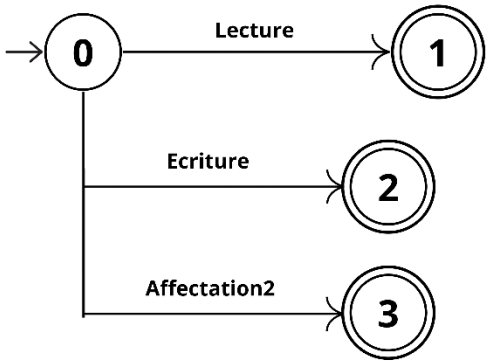
Bloc_Instruction = [Instruction1 SEMICOLON [Bloc_Instruction] | Instruction2 [Bloc_Instruction]]

```
def p_bloc_instruction(p):
    '''Bloc_Instruction : Instruction1 SEMICOLON Bloc_Instruction
                        | Instruction2 Bloc_Instruction
                        | Instruction1 SEMICOLON
                        | Instruction2
                        | Empty'''
    if len(p) == 4:
        p[0] = (p[1], p[2], p[3])
    elif len(p) == 3:
        p[0] = (p[1], p[2])
    else:
        p[0] = p[1]
```



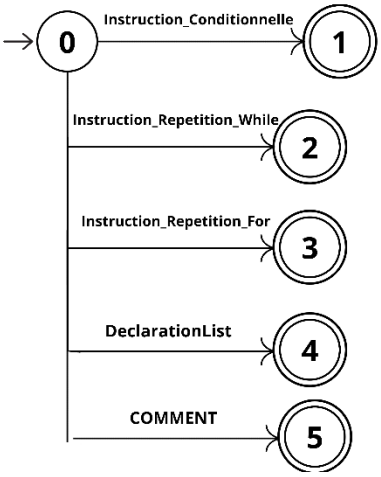
Instruction1 = Lecture | Ecriture | Affectation2

```
def p_instruction1(p):
    '''Instruction1 : Lecture
                    | Ecriture
                    | Affectation2'''
    p[0] = p[1]
```



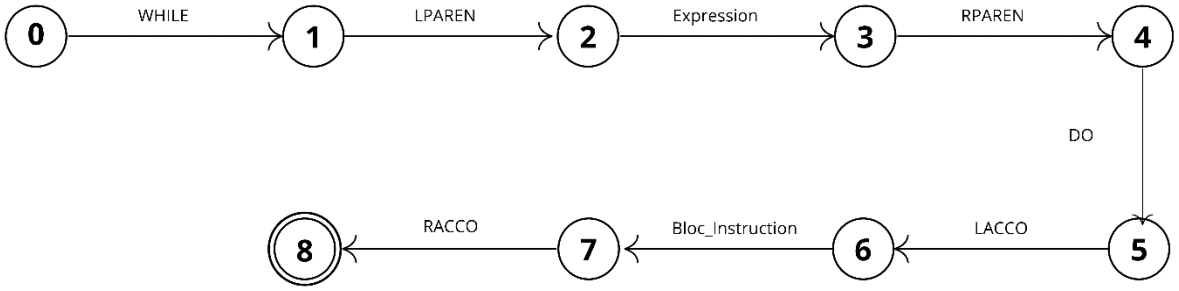
Instruction2 = Instruction_Conditionnelle | Instruction_Repetition_While | Instruction_Repetition+For | DeclarationList | COMMENT

```
def p_instruction2(p):
    '''Instruction2 : Instruction_Conditionnelle
                    | Instruction_Repetition_While
                    | Instruction_Repetition_For
                    | DeclarationList
                    | COMMENT'''
    p[0] = p[1]
```



Instruction_Repetition_While = WHILE LPAREN Expression RPAREN DO LACCO Bloc_Instruction RACCO

```
def p_instruction_repetition_while(p):
    '''Instruction_Repetition_While : WHILE LPAREN Expression RPAREN DO LACCO Bloc_Instruction RACCO'''

    p[0] = (p[1], p[2], p[3],p[4],p[5],p[6])
```

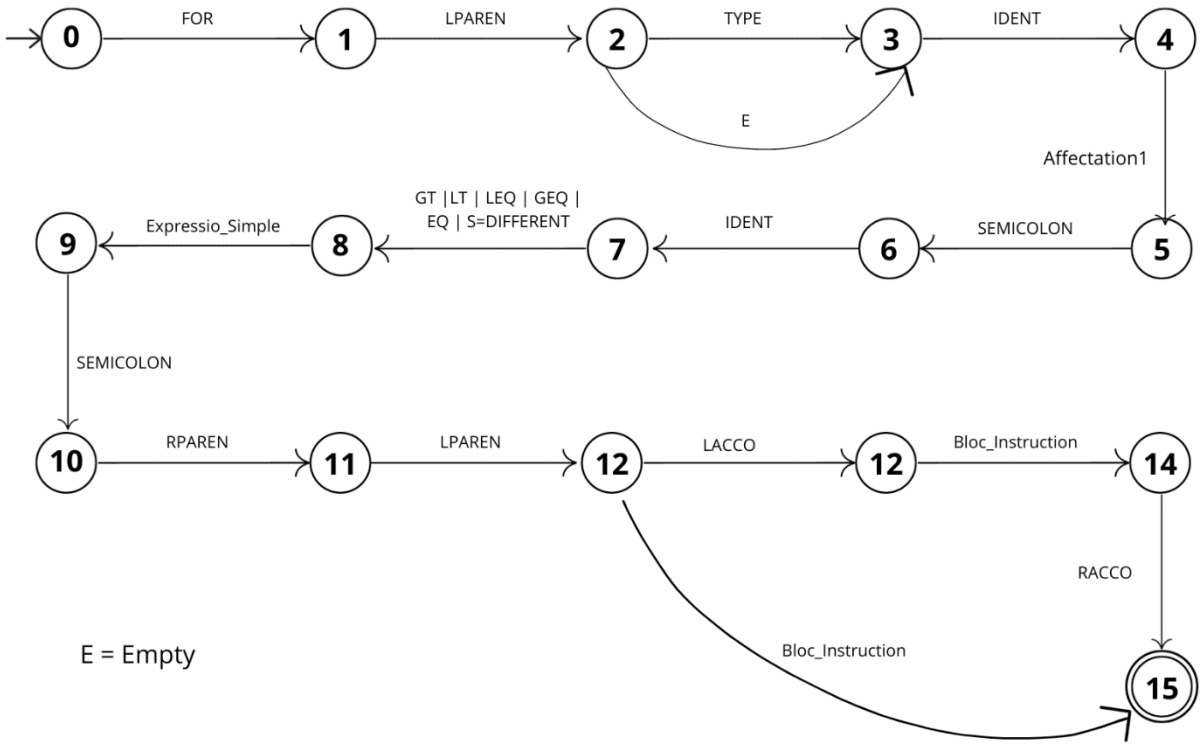


Instruction_Repetition_For = FOR LPAREN [TYPE] IDENT Affectation1 SEMICOLON IDENT (GT | LEQ | LT | GEQ | DIFFERENT | EQ) Expression_Simple SEMICOLON Affectation2 RPAREN (LACCO Bloc_Instruction RACCO | Bloc_Instruction)

```
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
```

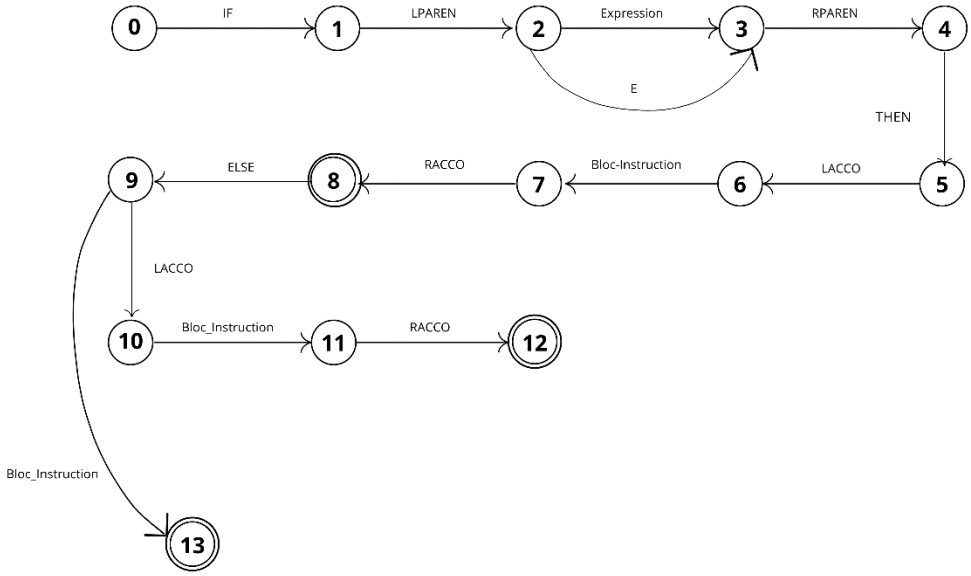
```
                                | FOR LPAREN TYPE IDENT Affectation1
SEMICOLON IDENT DIFFERENT Expression_Simple SEMICOLON Affectation2 RPAREN
LACCO Bloc_Instruction RACCO
                                | FOR LPAREN TYPE IDENT Affectation1
SEMICOLON IDENT EQ Expression_Simple SEMICOLON Affectation2 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
```

```
                                | 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])
```



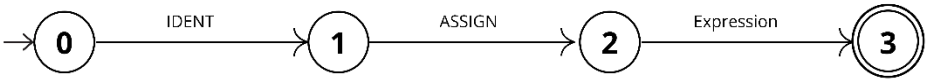
Instruction_Conditionnelle = IF LPAREN Expression RPAREN THEN (LACCO Bloc_Instruction RACCO | Bloc_Instruction)]

```
def p_instruction_conditionnelle(p):
    '''Instruction_Conditionnelle : IF LPAREN Expression RPAREN THEN LACCO Bloc_Instruction RACCO ELSE LACCO Bloc_Instruction RACCO
                                  | IF LPAREN Expression RPAREN THEN LACCO Bloc_Instruction RACCO ELSE Bloc_Instruction
                                  | IF LPAREN Expression RPAREN THEN Bloc_Instruction ELSE LACCO Bloc_Instruction RACCO
                                  | IF LPAREN Expression RPAREN THEN Bloc_Instruction ELSE Bloc_Instruction
                                  | IF LPAREN Expression RPAREN THEN LACCO Bloc_Instruction RACCO
                                  | IF LPAREN Expression RPAREN THEN Bloc_Instruction'''

    if len(p) == 7:
        p[0] = (p[1], p[2], p[3], p[4], p[5],p[6])
    else:
        p[0] = (p[1], p[2], p[3], p[4], p[5] ,p[6], p[7],p[8])
```



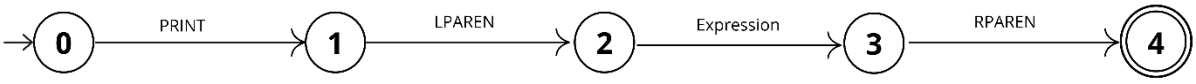
Affectation2 = IDENT ASSIGN Expression

```
def p_affectation2(p):
    '''Affectation2 : IDENT ASSIGN Expression'''
    p[0] = (p[1], p[2], p[3])
```



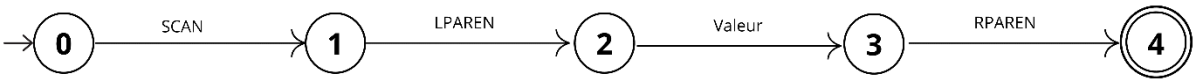
Lecture = PRINT LPAREN Expression RPAREN

```
def p_lecture(p):
    '''Lecture : PRINT LPAREN Expression RPAREN'''
    p[0] = (p[1], p[2], p[3], p[4])
```



Ecriture = SCAN LPAREN Valeur RPAREN

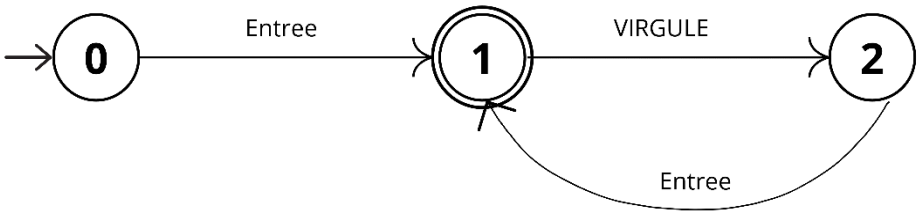
```
def p_ecriture(p):
    '''Ecriture : SCAN LPAREN Valeur RPAREN'''
    p[0] = (p[1], p[2], p[3], p[4])
```



Valeur = Entree { VIRGULE Entree }

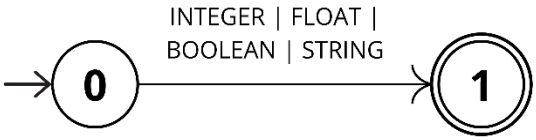
```
def p_valeur(p):
    '''Valeur : Entree EntreeList
              | Entree'''
    if len(p) == 3:
        p[0] = (p[1], p[2])
    else:
        p[0] = p[1]
```

```
def p_entreelist(p):
    '''EntreeList : VIRGULE Entree EntreeList
                  | VIRGULE Entree'''

    if len(p) == 4:
        p[0] = (p[1], p[2], p[3])
    else:
        p[0] = (p[1], p[2])
```



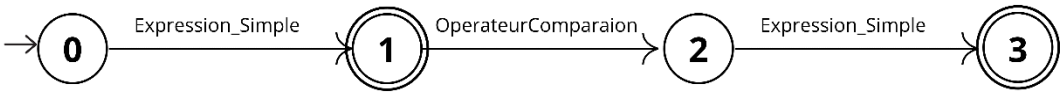
Entree = STRING | BOOLEAN | FLOAT | INTEGER

```
def p_entree(p):
    '''Entree : STRING
              | BOOLEAN
              | FLOAT
              | INTEGER'''

    p[0] = p[1]
```



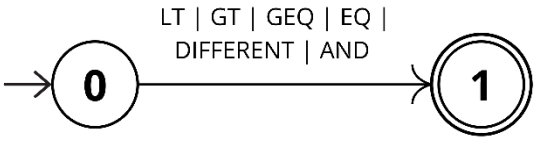
Expression = Expression_Simple [OperateurComparaison Expression_Simple]

```python
def p_expression(p):
    '''Expression : Expression_Simple OperateurComparaison Expression_Simple
                  | Expression_Simple'''
    if len(p) == 4:
        p[0] = (p[1], p[2], p[3])
    else:
        p[0] = p[1]
```



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

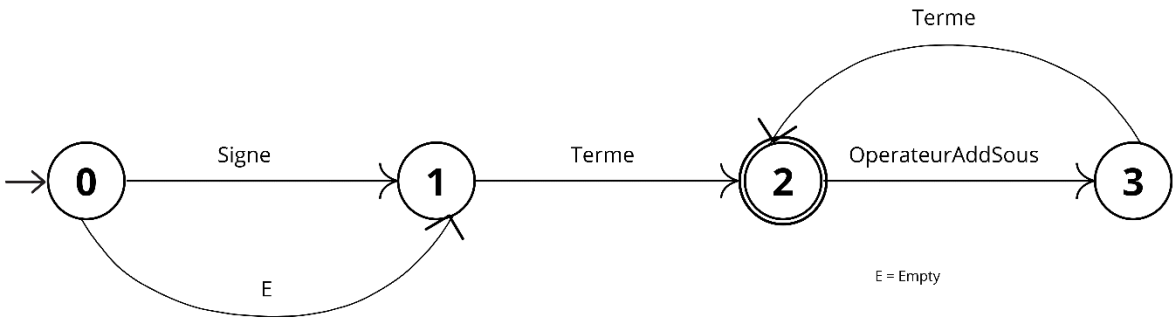
```python
def p_operateur_comparaison(p):
    '''OperateurComparaison : LT
                            | GT
                            | LEQ
                            | GEQ
                            | EQ
                            | DIFFERENT
                            | AND   '''
    p[0] = p[1]
```



Expression_Simple = [Signe] Terme { OperateurAddSous Terme }

```python
def p_expression_simple(p):
    '''Expression_Simple : signe Terme OperateurAddSousTermeList
                         | signe Terme
                         | Terme OperateurAddSousTermeList
                         | Terme'''
    if len(p) == 4:
        p[0] = (p[1], p[2], p[3])
    elif len(p) == 3:
        p[0] = (p[1], p[2])
    else:
        p[0] = p[1]
```

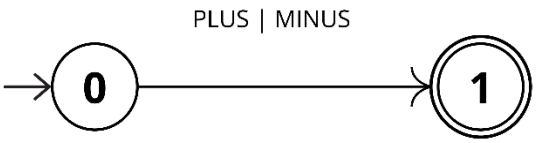
XXXXXXXX



Signe = PLUS | MINUS

OperateurAddSous = PLUS | MINUS

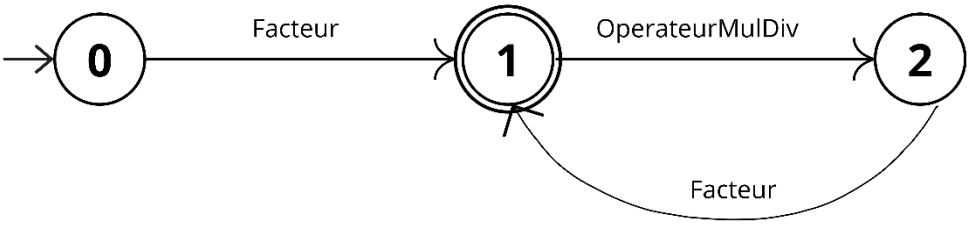
```
def p_signe(p):
    '''signe : PLUS
             | MINUS'''
    p[0] = p[1]

def p_operateur_add_sous(p):
    '''OperateurAddSous : PLUS
                        | MINUS'''
    p[0] = p[1]
```



Terme = Facteur { OperateurMulDiv Facteur }

```
def p_terme(p):
    '''Terme : Facteur OperateurMulDivFacteurList
             | Facteur'''
    if len(p) == 3:
        p[0] = (p[1], p[2])
    else:
        p[0] = p[1]
```

```
def p_operateurMulDivFacteurList(p):
    '''OperateurMulDivFacteurList : OperateurMulDiv Facteur OperateurMulDivFacteurList
                                  | OperateurMulDiv Facteur'''
    if len(p) == 4:
        p[0] = (p[1], p[2], p[3])
    else:
        p[0] = (p[1], p[2])
```

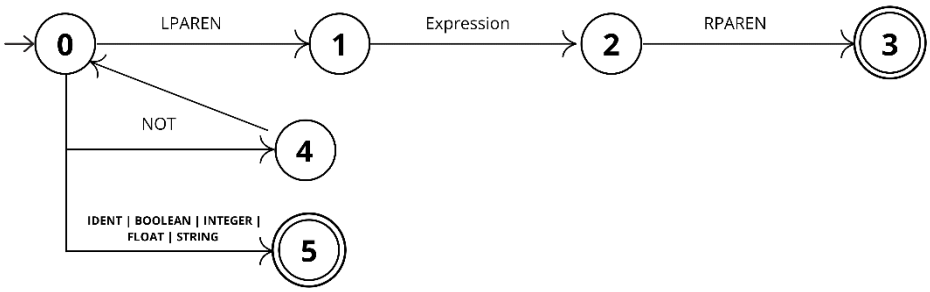


OperateurMulDiv = TIMES | DIVIDE

```
def p_operateur_mul_div(p):
    '''OperateurMulDiv : TIMES
                       | DIVIDE'''
    p[0] = p[1]
```

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

```
def p_facteur(p):
    '''Facteur : LPAREN Expression RPAREN
               | NOT Facteur
               | IDENT
               | BOOLEAN
               | INTEGER
               | FLOAT
               | STRING'''
    if len(p) == 4:
        p[0] = (p[1], p[2], p[3])
    elif len(p) == 3:
        p[0] = (p[1], p[2])
    else:
        p[0] = p[1]
```



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.

```
def p_empty(p):
    'Empty :'
    pass
```

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)`.

```
# Error rule for syntax errors
def p_error(p):
    if p:
        print(f"Syntax error at line {p.lineno}, pos {p.lexpos}: Unexpected token '{p.value}' in rule '{p.type}'")
        print(" ")
        print("La sequence d'instructions est syntaxiquement incorrecte")
        print("Les instructions ne sont pas reconnus par le PARSER")
        print(" ")
        sys.exit(1)
    else:
        print("Syntax error: Unexpected end of input")
        print(" ")
        print("La sequence d'instructions est syntaxiquement incorrecte")
        print("Les instructions ne sont pas reconnus par le PARSER")
        print(" ")
        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.

```
# Build the lexer
lexer = lex.lex()
# Test the lexer with a sample input
data = """
//djhthdh;
int a=5;
while(a>5) do {
  print(a);}
for(i=1;i<=10;i=i+1) {
    i=i+1;
}
if(a==5) then {
    int a=2;
}
//djhthdh;

"""
print("")
print("")
print("LEXER")
print("")
print("")
lexer.input(data)
while True:
    tok = lexer.token()
    if not tok:
        break  # No more input
    tok.lexpos = lexer_pos_in_line
    lexer_pos_in_line += len(tok.value)
    print(f"{tok.type} at line {tok.lineno}, pos {tok.lexpos}: {tok.value}")
```

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.

```
420    print("La sequence d'instructions est lexicalement correcte")
421    print("Tous les lexemes sont recconnus par le LEXER")
422    print("")
423    print("")
424
425    print("PARSER")
426    print("")
427    print("")
428
429    # Build the parser
430    parser = yacc.yacc()
431    result = parser.parse(data,lexer=lexer)
432    print(result)
433
434    print("")
435    print("")
436    print("La sequence d'instructions est syntaxiquement correcte")
437    print("Tous les instructionss sont reconnus par le PARSER")
438    print("")
439    print("")
440
441
```

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.