



## Rapport de TP

---

### L'analyseur lexical

Réalisé par : Erraji Mouad  
Ahmed Chakir Alaoui

Encadré par : Mr. Anwar

## Sommaire

Introduction.....	3
Qu'est ce que L'analyse lexicale ?.....	6
Implémentation de l'analyseur lexical .....	7
Utilisation d'une matrice décrivant l'automate de l'analyseur lexical avec un programme l'exploitant en Python : .....	7
1.Légende : .....	7
2.Grammaire: .....	7
3.Automates & expressions régulières : .....	8
4.Automate de l'analyseur lexical : .....	10
5.Matrice de transition : .....	11
I – Structure des données : .....	12
1 – Matrice Creuse : .....	12
2- Lexème : .....	12
3- Table de symboles: .....	12
II – Fonctions Utiles : .....	13
4- Fonction de recherche dans la matrice : .....	13
5 – Fonction de correspondance : .....	13
6 - Fonction de l'état suivant : .....	15
7 – Fonction de l'état finale : .....	15
8 - Fonction de l'unité lexicale .....	16
9 – Fonction de Hashage : .....	16
10 – Fonction de remplissage : .....	17
11 – Fonction d'affichage de la table de symbole : .....	17
III – Programme Principale : .....	18
Conclusion .....	20

# Introduction

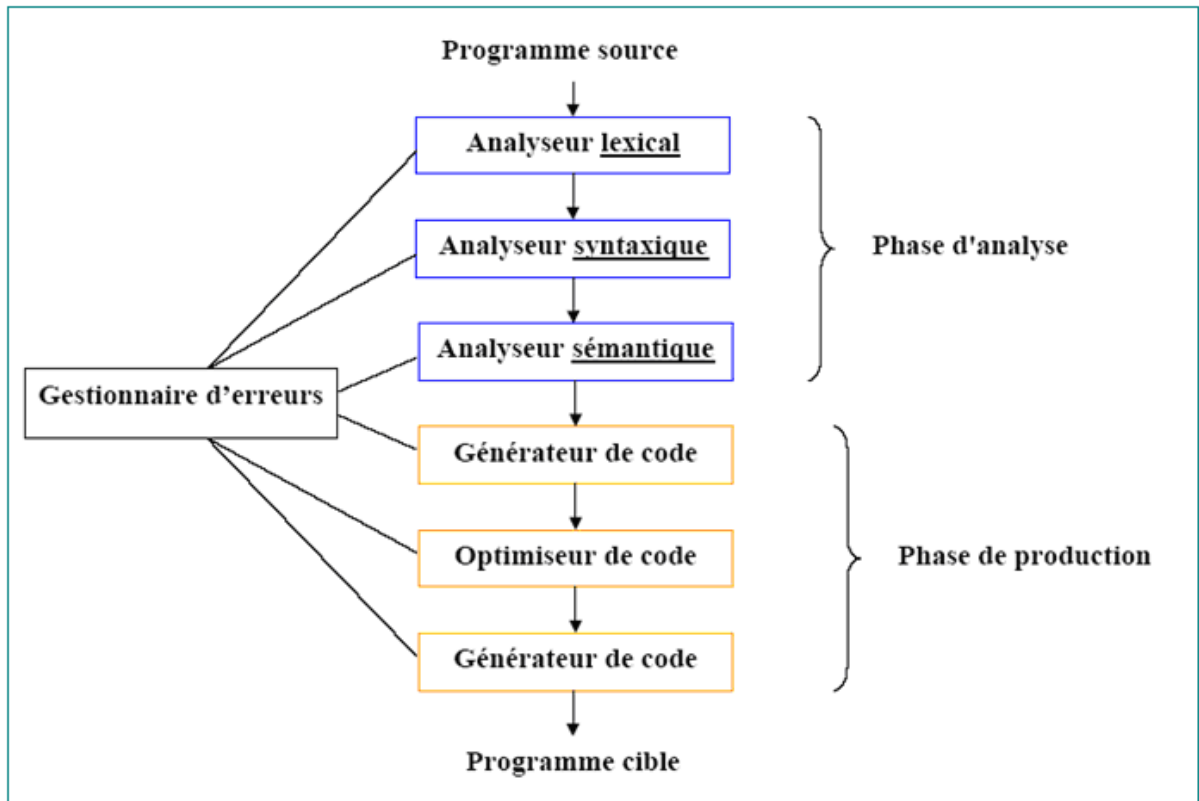
Le compilateur est un outil qui est utilisé par tout programmeur de nos jours. En effet, c'est un outil présent dans la plupart des applications basées sur la programmation. Aussi l'étude d'un tel outil est-elle fondamentale dans le cursus d'un informaticien et c'est dans cette optique que nous nous intéressons à comprendre le fonctionnement d'un tel compilateur, à étudier ses caractéristiques, les principaux services qu'il rend et les différentes étapes nécessaires afin de le réaliser.

Un compilateur est un programme informatique qui traduit un fichier source d'un langage donné en langage assembleur (cible). C'est ce langage que la machine exécute réellement tout en signalant les éventuelles erreurs présentes dans le programme source.

La réalisation d'un tel compilateur passe par plusieurs étapes :

- ✓ *L'analyse lexicale* qui consiste en un découpage du programme en lexèmes<sup>1</sup>.
- ✓ *L'analyse syntaxique* qui analyse la syntaxe du programme et corrige les éventuelles erreurs syntaxiques.
- ✓ L'analyse sémantique qui s'occupe de l'analyse des différentes structures de données.
- ✓ La construction d'un arbre syntaxique qui consiste en la construction en mémoire d'un arbre représentant le code à compiler.
- ✓ La génération du code qui se fait par un parcours de l'arbre en mémoire.

La figure ci-dessous schématise les différentes étapes citées ci-dessus :



Toutefois, la compilation peut être scindée en deux parties essentielles :

- ✓ La partie analyse qui partitionne le programme source en ses constituants et en crée une représentation intermédiaire.
- ✓ La partie synthèse qui construit le programme cible désiré à partir de cette représentation intermédiaire.

Nous présenterons, dans ce rapport, le travail que nous avons réalisé pendant trois mois et qui a consisté en l'implémentation d'un analyseur lexical qui analyse une chaîne donnée par l'utilisateur et en extrait les différentes unités lexicales ainsi que les éventuelles erreurs présentes sur cette même chaîne. Cet analyseur lexical représente, pour l'instant, une coquille vide sur laquelle bouclera l'analyseur syntaxique que nous tenterons de réaliser dans les mois à venir.

La programmation d'un tel analyseur est passée par plusieurs étapes. Aussi tenterons-nous d'éclaircir tout au long de ce rapport le plan de travail que nous avons suivi.

## Qu'est ce que L'analyse lexicale ?

L'analyse lexicale est une analyse microscopique des éléments formant un programme.

### Principales tâches :

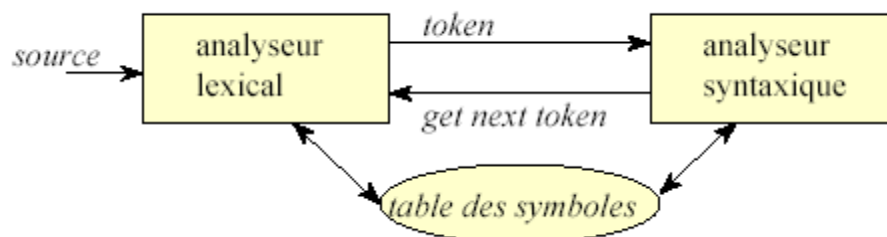
- Lire les caractères du programme source
- Produire un flot de mots appelés lexèmes (mots du programme source ayant une unité lexicale).
- Reconnaître la classe ou l'unité lexicale de chaque lexème en se basant sur des automates.

### Interaction entre l'analyseur lexical et l'analyseur syntaxique :

L'analyseur syntaxique a une vue globale sur tous les modules du compilateur. Il peut même être assimilé à un chef d'orchestre puisqu'il interagit parallèlement avec ceux-ci. En effet, il récupère les unités lexicales isolées par l'analyseur lexical et les transmet à l'analyseur sémantique avant de générer le code et ceci se faisant suite de symboles par suite de symboles.

Par ailleurs, l'analyseur lexical élimine tout ce qui n'est pas utile à l'analyseur syntaxique notamment en ignorant les commentaires et les blancs.

La figure ci-dessus illustre les échanges décrits ci-dessus:



Toutefois, l'analyseur lexical peut être écrit soit :

- ✓ Par une table décrivant l'automate et un programme exploitant cette table,
- ✓ Par un générateur d'analyseurs : Flex.




## Implémentation de l'analyseur lexical

### Utilisation d'une matrice décrivant l'automate de l'analyseur lexical avec un programme l'exploitant en Python :

L'utilisation d'une matrice de transition décrivant l'automate d'un analyseur lexical est l'une des approches que l'on a abordées afin de le réaliser.

Dans un premier temps, nous avons établi les différents automates nécessaires pour l'implémentation de la matrice de transition.

#### 1.Légende :

	Etat initial
	Etat intermédiaire
	Etat final
Tout caractère	T
Lettre	L
Chiffre	C
V1	{A B ... Z a b ... z}
V2	{0 1 ... 9}
V3	{1 2 ... 9}

#### 2.Grammaire:

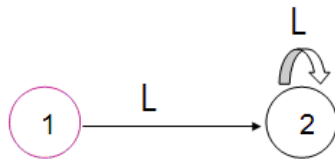
$G = \{V1, V2, V3, +, -, /, *, (, ), , <, >, =, :, , ;, ., "\}$

- V1 appartient à {a,.....z, A,.....Z}
- V2 appartient à {0,.....,9}
- V3 appartient à {1,.....,9}

### 3. Automates & expressions régulières :

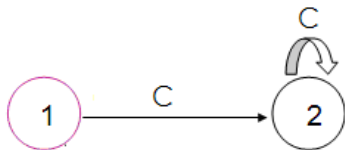
- Identificateur : Suite de caractères alphabétiques.

$$ER = (V1.V1^*)$$

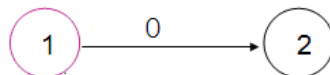


- Constante numérique : Suite de chiffre ne commençant pas par 0.

$$ER = (V3.V2^*)$$

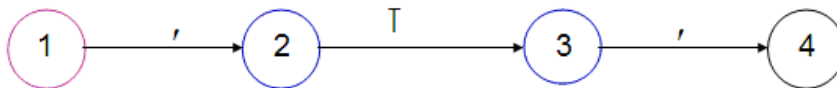


- Constante nulle :



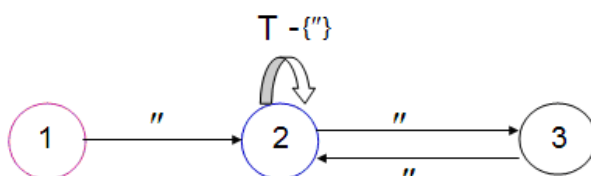
- Constante caractère : Un caractère encadré par deux côtes

$$ER = ('T')$$



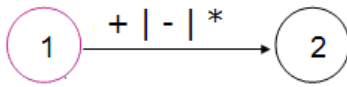
- Chaîne de caractères : Suite de caractères encadrés par " "

$$ER = (" \{T \setminus \{ " \} \}^* "). (" \{T \setminus \{ " \} \}^* ")^*$$

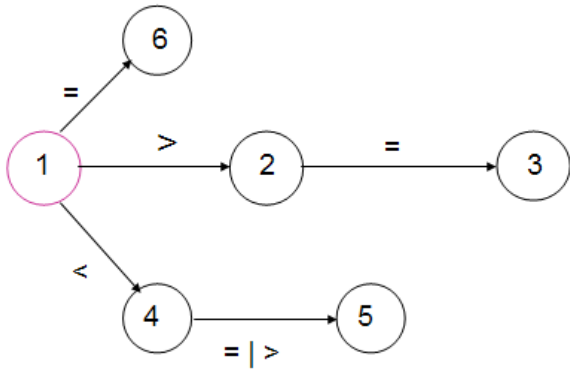




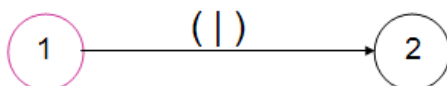
- Opérateur arithmétique : ER = (+ | - | \*)



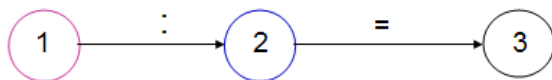
- Opérateur de relation : ER = ((>.=) | (<. (= | > )) | =)



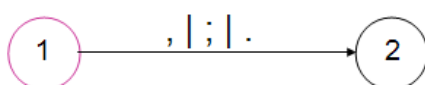
- Parenthèse : ER = ( | )



- Symbole d'affectation : ER = ( :=)

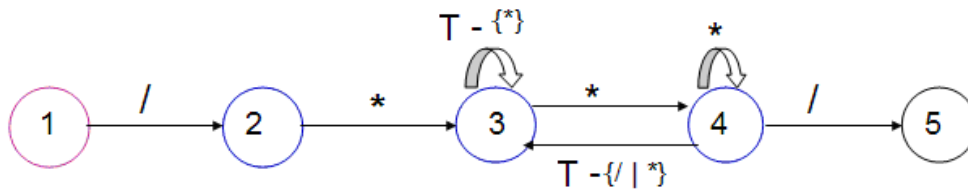


- Ponctuation : ER = (, | ; | .)

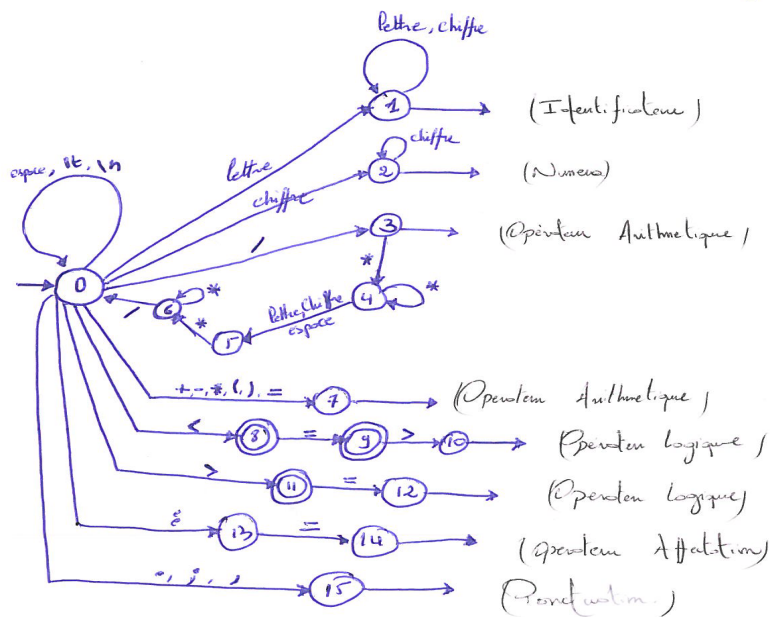


- Commentaire : Un commentaire est une chaîne de caractères encadrée par /\*\*/

$$ER = (/^*)[T \setminus \{*\} \mid (^*)^+ T \setminus \{/|\} ]^*(*/)$$



#### 4. Automate de l'analyseur lexical :



état  
initiale  $\rightarrow$  0

état  
finale  $\rightarrow$  1, 2, 3, 4, 5, 7, 9, 10, 12, 14, 15

## 5. Matrice de transition :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	1	2	3	7	7	7	7	7	7	8	11	13	15	15	15	0	0	0
1	1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	5	5	-1	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5	-1	-1
5	5	5	-1	6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5	-1	-1
6	-1	-1	0	6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	-1	-1	-1	-1	-1	-1	-1	-1	9	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	10	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	-1	-1	-1	-1	-1	-1	-1	-1	12	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
13	-1	-1	-1	-1	-1	-1	-1	-1	14	-1	-1	-1	-1	-1	-1	-1	-1	-1
14	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

### Numéro Caractère correspondant Colonne

0	Lettre
1	Chiffre
2	/
3	*
4	+
5	-
6	(
7	)
8	=
9	<
10	>
11	:
12	.
13	,
14	;
15	Espace
16	\n
17	\t

Les commentaires sont dans la forme suivante :

/\* Un commentaire \*/

(Le nombre d'étoiles peut varier entre 1 et plusieurs)

## I - Structure des données :

### 1 - Matrice Creuse :

Pour des raisons d'optimisation, nous avons modélisé notre matrice a l'aide d'une matrice creuse :

```
#Matrice creuse de l'Automate

A = [1, 2, 3, 7, 7, 7, 7, 7, 7, 8, 11, 13, 15, 15, 15, 0,
0, 0, 1, 1, 2, 4, 5, 5, 4, 5, 5, 5, 6, 5, 0, 6, 9, 10,
12, 14, 0, 0]

AI = [18, 20, 21, 22, 26, 30, 32, 32, 33, 34, 34, 35, 35,
36, 36, 36]

AJ = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 0, 1, 1, 3, 0, 1, 3, 15, 0, 1, 3, 15, 2, 3,
8, 10, 8, 8]
```

### 2- Lexème :

A fin de représenter les lexème, nous avons utilisé un type de structure existant dans Python : les 'List' . C'est un genre de tableau qui peut contenir différent type de données, équivalent a l'ArrayList en java.

```
#Structure Lexème

' ' ' lexème = [mot, unite lexicale] ' ' '

lexeme = [None, None]
```

### 3- Table de symboles:

La table de symboles compte a elle est représenté en utilisant une 'List' de 'List' de lexème. Chaque 'List' contiendra une 'List' de lexème ayant le même hash-code. Nous allons initialisés la table avec des données 'None', pour fixer sa taille. Et puis nous allons y mettre tous les mot clés, après avoir défini la fonction permettant l'ajout.

```
#Structure Table de Symbole
'''
tableSymbole = [[lex1, lex2], [lex]]
lex1 et lex2 ont le même hash Code
>>>Initialisation:
```

```
'''
tableSymbole = [ [None, None]] ]
for i in range (39):
    tableSymbole.append([lex,])
```

## II – Fonctions Utiles :

### 4- Fonction de recherche dans la matrice :

```
#####
##

#Fonction de recherche dans la Matrice
def rechercher (ligne, colonne):
    trouve = 0
    if ligne == 0:
        up = 0
        down = AI[ligne] - 1
    else:
        up = AI[ligne - 1]
        down = AI[ligne] - 1
    for x in range(up, down +1):
        if AJ[x] == colonne:
            valeur = x
            trouve = 1
    if trouve == 0:
        return -1
    else:
        return A[valeur]

#####
##
```

### 5 – Fonction de correspondance :

Cette fonction permet de correspondre a chaque numéro de colonne , le caractère auquel il réfère. Pour ce faire, nous avons utilisé le code ASCII de chaque caractère, qui est généré grace a une fonction de Python : 'ord'.

```
#####
##

#Fonction return numero de colonne de chaque caractere
```

```

def caractereAcolonne (caractere):
    '''
        a un caractere donné, retourne la colonne
        correspondante
    '''
    #lettre
    if ord(caractere) in range(65, 91) or ord(caractere)
in range(97, 123):
        return 0
    #chiffre
    elif ord(caractere) in range(49, 58):
        return 1
    #/
    elif ord(caractere) == 47:
        return 2
    #*
    elif ord(caractere) == 42:
        return 3
    #+
    elif ord(caractere) == 43:
        return 4
    #-
    elif ord(caractere) == 45:
        return 5
    #(
    elif ord(caractere) == 40:
        return 6
    #)
    elif ord(caractere) == 41:
        return 7
    #=
    elif ord(caractere) == 61:
        return 8
    #<
    elif ord(caractere) == 60:
        return 9
    #>
    elif ord(caractere) == 62:
        return 10
    #:
    elif ord(caractere) == 58:
        return 11
    #.
    elif ord(caractere) == 46:
        return 12
    #,
    elif ord(caractere) == 44:
        return 13
    #;
    elif ord(caractere) == 59:
        return 14

```

```

#espace
elif ord(caractere) == 32:
    return 15
#\t
elif ord(caractere) == 9:
    return 16
#\n
elif ord(caractere) == 10:
    #Caractere non reconnue
else:
    return -1

```

## 6 - Fonction de l'etat suivant :

Fonction qui a un etat donnees, et un caractère de transition donné , nous indique l'etat suivant.

```

#####
#
#Fonction return Etat Suivant
def etatSuivant(etat, caractere):
    ligne = etat
    colonne = caractereAcolonne(caractere)
    return rechercher(ligne, colonne)

#####
#

```

## 7 - Fonction de l'etat finale :

Fonction bouleene qui nous indique si un etat est finale

```

#####
#Fonction is Etat Final
def estFinale(etat):
    '''
    return boolean : si l'etat est final ou pas
    '''
    final = False
    if etat == 1 or etat == 2 or etat == 3 or etat == 7
or etat == 8 or
        etat == 9 or etat == 10 or etat == 11 or etat ==
12 or etat == 14 or etat == 15:
        final = True
    return final

```

## 8 - Fonction de l'unité lexicale

Fonction qui nous indique qu'elle est l'unité lexicale d'un mot donnés, selon l'état finale par le quel il est sortie.

```
#####  
##  
  
#Fonction return UL  
def uniteLexicale(etatFinale):  
    '''  
    retourne l'unité lexical selon l'état final  
    sa depend d'ou on est sortit dans l'automate  
    '''  
    ul = -1  
    if etatFinale == 1:  
        ul = 'Identificateur'  
  
    if etatFinale == 2:  
        ul = 'Nombre'  
  
    if etatFinale == 8 or etatFinale == 9 or etatFinale  
    == 10 or etatFinale == 11 or etatFinale == 12 or  
    etatFinale == 14:  
        ul = 'Opérateur Logique'  
    if etatFinale == 3 or etatFinale == 7 :  
        ul = 'Opérateur Arithmétique'  
    if etatFinale == 14:  
        ul = 'Opérateur Affectation'  
  
    return ul  
  
#####  
##
```

## 9 - Fonction de Hashage :

Fonction qui pour un mot données renvoie un hash-code. Ce code est calculé grace a un fonction de hashage : La somme des codes ASCII de chaque caractère , multiplié par leur position , le tout modulo 40.

```
#Fonction Hashage du lexème  
def hashCode(mot):  
    somme = 0  
    i = 1  
    for lettre in mot:  
        somme += i * ord(lettre)
```



```

        i += 1
    h = somme % 40
    return h

```

## 10 - Fonction de remplissage :

Fonction qui permet d'ajouter dans la table des symboles. Elle est d'abord initialisé avec des mots clés, puis au fur et à mesure que l'on analyse les expressions, la fonction ajoute à la table de symboles les identificateurs, sans duplication bien sûr.

```

#####
##

#Fonction Remplir Table des Symboles
def ajouterTableSymbole(lexeme):
    mot = lexeme[0]
    ul = lexeme[1]
    h = hashCode(mot)
    if lexeme not in tableSymbole[h] and (ul ==
'Identificateur' or ul == 'Mot Clé'):
        tableSymbole[h].append(lexeme)
        if tableSymbole[h][0] == [None, None]:
            tableSymbole[h].pop(0)

#Mot Clé
motCle = ['si', 'finsi', 'fin', 'alors', 'sinon',
'debut', 'tantque', 'et', 'ou']
for mot in motCle:
    lex = [mot, 'Mot Clé']
    ajouterTableSymbole(lex)

#####
##

```

## 11 - Fonction d'affichage de la table de symbole :

Finalement, afin de visualiser notre table de symboles et ce qu'elle contient, nous avons réalisé une fonction qui répond à ce besoin.

```

#####
##

#Fonction Afficher Table des Symboles
def afficher(table):
    Print

```

```

print ' _____ '
print 'mot', '\t', '\t', 'unite lexicale'
print ' _____ ', '\t', '\t', ' _____ '
Print
for h in table:
    for lex in h:
        if lex[0] != None and lex[1] !=
None:
            print
lex[0], '\t', '\t', lex[1]

```

### III – Programme Principale :

Et voilà enfin notre analyseur prêt à l'emploi, il vous suffit maintenant de copier/coller l'expression que vous souhaitez analyser après la requête vous le demandant, et notre programme se chargera de reconnaître chaque mot, de vous indiquer son unité lexicale, de remplir la table de symbole, et de vous l'afficher.

```

#Fonction Principale
def main():

    expression = raw_input('Entrez votre expression = ')
    expression += ' '
    etat = 0
    mot = ''
    i = 0
    Print
    while i < len(expression) :

        etatPrecedent = etat
        etat = etatSuivant(etat, expression[i])
        mot += expression[i]

        if etat == 0:
            mot = ''

        if estFinale(etatPrecedent) and (etat == -1 or
expression[i] == expression[-1]):

            mot = mot[:-1]
            if mot in motCle:
                ul = 'Mot Clé'
            else :
                ul = uniteLexicale(etatPrecedent)

            lexeme = [mot, ul]
            ajouterTableSymbole(lexeme)
            print mot + '\t', '(' + ul + ')' '

```

```

        mot = ''
        etat = 0
        i -= 1
    if not estFinale(etatPrecedent) and etat == -1 :
        mot = mot[:-1]
        ul = 'Mot Non Reconnu'
        print mot + '\t','(' + ul + ')'
        mot = ''
        etat = 0
        if caractereAcolonne(expression[i]) != -1 :
            i -= 1
    i += 1
afficher(tableSymbole)

```

## Conclusion

Ainsi, nous avons pu atteindre l'objectif que nous nous étions fixés. En effet, nous avons pu réaliser un petit analyseur lexical en langage Python.

La conception de ce module est, certes, très importante mais concevoir un compilateur, à proprement dit, requiert que nous nous penchions sur la conception d'un nouveau module qui n'est autre que l'analyseur syntaxique .

Ainsi, ce travail nous a permis de consolider nos connaissances en programmation et d'acquérir un certain savoir en ce qui concerne la théorie des langages et le mode opératoire d'un compilateur.

