**Université Sidi Mohamed Ben Abdellah**
**Faculté des Sciences Dhar El Mahraz - Fès**

**2019/2020**

# Intelligence Artificiels

**Sur**

# *Tableau de Verité*
# *Code par Python*

Fait par :

*REDA RAFI*

rida.bleuman@gmail.com

# Table de Matière

# Introduction

## *La logique dans la programmation*

### *Définitions*

*La logique est extrêmement importante à la fois dans le matériel et les logiciels informatiques.*
*Nous commencerons ici par les aspects logiciels de la logique impliqués dans la programmation.*
*Plus tard, nous montrerons brièvement certains aspects matériels des portes impliquées dans l'architecture informatique.*
*La logique implique des conditions dans pratiquement toutes les constructions Choix et Loop (formes If et While).*
*La logique se produit également dans les assertions, les conditions préalables, les conditions de publication, les invariants et même les commentaires.*

*Une table de vérité est une table mathématique utilisée en logique classique — en particulier le calcul propositionnel classique et l'algèbre de Boole — pour représenter de manière sémantique des expressions logiques et calculer la valeur de leur fonction relativement à chacun de leurs arguments fonctionnels (chaque combinaison de valeur assumée par leurs variables logiques). Les tables de vérité peuvent être utilisées en particulier pour dire si une proposition est vraie pour toutes les valeurs légitimement imputées, c'est-à-dire : si une proposition est « logiquement valide ».*

# *Exemples de base*

Les quatre tables de vérité présentées ci-dessous permettent de définir les connecteurs logiques et, ou, ou exclusif et implication en mathématiques, ou les portes logiques correspondantes en électronique.

Par exemple, en langage électronique, nous devons avoir les deux entrées à 1 pour que la sortie de la porte logique ET soit activée ; alors que la porte logique OU n'a besoin que d'une des entrées à 1 pour afficher un 1 à la sortie ; ou encore nous devons avoir a et b ayant la même entrée ou qu'a soit FAUX et b soit VRAI pour avoir un 1 en sortie pour la porte logique de l'implication.

| Table de vérité de ET | | |
|---|---|---|
| a | b | a ET b |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| Table de vérité de OU | | |
|---|---|---|
| a | b | a OU b |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| Table de vérité de XOR (OU exclusif) | | |
|---|---|---|
| a | b | a XOR b |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| Table de vérité de l'implication | | |
|---|---|---|
| a | b | $a \Rightarrow b$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# *Exemple composé*

| Table de vérité de $a \wedge (b \vee c)$ avec calcul intermédiaire de $b \vee c$ | | | | |
|---|---|---|---|---|
| a | b | c | b ∨ c | a ∧ (b ∨ c) |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Le '∧' se lit et, le '∨' se lit ou.

Les 3 premières colonnes de ce tableau fournissent les valeurs de vérité de a, b et c (par ex. des propositions logiques en mathématiques, ou des états logiques en électronique).

La 4ème colonne fournit les valeurs de vérité de : b ou c.

La 5ème colonne fournit les valeurs de vérité de : a et (b ou c).

# Algorithme

## *Implémentation du tableau de vérité*

**Présentation d'objet**

*En effet cette partie je vais vous mon algorithme étape par étape ; tableau de vérité coder avec python.*
*Or que le code est une suite de fonction, chaque fonction traduit un ordre bien détailler par la suit en vas voir tous ces derniers.*

**Présentation des fonctions**

```
class myStack:
    def __init__(self):
        self.stack = []
    def push(self, x):
        self.stack.append(x)
    def size(self):
        return len(self.stack)
    def empty(self):
        return self.size() == 0
    def top(self):
        return self.stack[-1]
    def pop(self):
        res = self.top()
        self.stack.pop()
        return res
```

```
def Priority(c : str):
# retourner l'ordre de priorité de l'opérateur
# detection de symbole
    if (c == '('):
        return 0
    elif (c == '>' or c == '~'):
        return 1
    elif (c == 'v' or c == '^'):
        return 2
    else:
        assert(c == '!')
        return 3
```

*Cette fonction « Priority » elle convertie chaque symbole trouver dans notre expression a un entier qui va m'aide a simplifier le travail ; vaut mieux travailler avec une transmission d'information par entier que par caractère*

```
def Oper(x : int, y : int , oper : str):
# Calculer l'expression (x oper y), laquelle 'oper' est dans [v, ^, <->, ->]
    if (oper == 'v'):
        return (x | y) #x v y
    elif (oper == '^'):
        return (x & y) #x ^ y
    elif (oper == '>'):
        return ((1 ^ y) | x) #x -> y
    else:
        assert(oper == '~')
        return (1 if (x == y) else 0) #x <-> y
```

*La fonction « Oper » retourne le résultat de chaque opération entre les paramètres*

```
def preprocess(expression : str):
    expression = re.sub(' ','', expression) # supprimer tout l'espace
#pour une programmation plus facile, j'ai converti tous les opérateurs mul
ti-caractères en un seul caractère
    expression = re.sub('<->', '~', expression)
    expression = re.sub('->', '>', expression)
#- et ! les deux représentent l'opérateur NOT
    expression = re.sub('-', '!', expression)
# + et v sont tous deux représentatifs de l'opérateur OR
    expression = re.sub('\+', 'v', expression)
#. et ^ les deux représentent l'opérateur AND
    expression = re.sub('\.', '^', expression)
# supprimer tout !! opérateur
    while (re.search('!!',expression) != None):
        expression = re.sub('!!', '', expression)
    return expression
```

*« Preprocess » supprimé tous les space ; puisque en va falloir lire l'expression donner par l'utilisateur qui est une List des caractères sa peut qu'il sois des caractère nulle « empty »*

*Et je convertie tous les multi-caractère a un seule unique pour faciliter la vision du code*

```
def GetVariable(expression : str):
    """Récupère toutes les variables apparues dans 'expression'"""
    SetVar = set()
    ListVar = []
    for c in expression:
        if ('a' <= c) and (c <= 'z') and (c != 'v'): #v is an operator
            if (c in SetVar):
                continue
            SetVar.add(c)
            ListVar.append(c)
    return ListVar

def GetRPN(expression : str):
# Utilisez l'algorithme de shunting-
yard pour obtenir la notation polonaise inverse de l'expression à partir de la notation Infix
    stack = myStack()
    RPN = myStack()
    for c in expression:
        if c == '(':
            stack.push(c)
        elif c == ')':
            while True:
                x = stack.pop()
                if (x != '('):
                    RPN.push(x)
                else:
                    break
        elif c in ['v','^','>','~','!']:
            while (not stack.empty()) and
                     Priority(c) <= Priority(stack.top()):
                RPN.push(stack.pop())
            stack.push(c)
        elif (c == '0' or c == '1' or (
             ('a' <= c) and (c <= 'z') and (c != 'v'))
             ):
            RPN.push(c)
    while (not stack.empty()):
        RPN.push(stack.pop())
    return RPN
```

*Cette fonction et aussi long mais il joue un grand rôle dans mon code car c'est avec elle que je détecte toutes les variables de ma table de vérité afin que je puisse par la suite savoir combien de relation j'aurais*
*Après je convertie l'expression originale a la notation polonaise afin que je sépare le groupement des sous-expressions c.-à-d. les sous-expressions entourer par la parenthèse a l'aide du RPN ;permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses.*

```
def Calculate(RPN : myStack, VariableValue: dict):
    """Calculer la valeur d'expression à partir de RPN, avec la valeur de
la variable est stockée dans 'VariableValue'"""
    res = myStack()
    for c in RPN.stack:
        if (c == '0'
               or c == '1'
               or (('a' <= c)
               and (c <= 'z')
               and (c != 'v'))
           ):
            #'v' est un opperateur
            res.push(VariableValue[c])
        elif (c == '!'):
            # not X, !X, -X
            res.push(1 ^ res.pop())
        else:
            assert(c == '^'
                   or c == 'v'
                   or c == '>'
                   or c == '~')
            x = res.pop()
            y = res.pop()
            res.push(Oper(x, y, c))
    assert(res.size() == 1)
    return res.pop()
```

*Après le RPN je reçois les sous expressions et je calcule leur logique*

```python
def WriteToConsole(result : list):
    for row in result:
        tmp = row.pop()
        for x in row:
            print("  ", x, end = "  |", sep = '')
        print("  ", tmp, sep = '')
        row.append(tmp)

def WriteToFile(result : list):
    csvFile = open('result.csv','w')
    writer = csv.writer(csvFile)
    writer.writerows(result)
    csvFile.close()
```

*Ces deux fonction simple*
*La premier affiche le résultat dans la console.*
*La deuxième affiche le résultat dans un fichier Excell nommer résult format .csv*

```python
def Solve(expression : str):
    result = []

    ListVariable = GetVariable(expression)
    ListVariable.append(expression)
    result.append(ListVariable.copy())
    ListVariable.pop()

    expression = preprocess(expression)
    RPN = GetRPN(expression)
    n = len(ListVariable)

  # force brute toutes les valeurs de toutes les variables
    for mask in range(2 ** n):
        VariableValue = {'0' : 0, '1' : 1}
        cur = []
        for i in range(n):
            VariableValue[ListVariable[i]] = (mask >> (n - i - 1) & 1)
            cur.append(mask >> (n - i - 1) & 1)
        cur.append(Calculate(RPN, VariableValue))
        result.append(cur)
    WriteToConsole(result)
    WriteToFile(result)
```

*Fonction solve c’est elle qui fait apple a tous les fonction precedent*

```
def main():
    file = open("file.txt","r")
    data=file.readlines()
    print("votre expression => ",data[4])
    # print(data[4])
    Solve(data[4])
    print("tableau stocker dans result.csv  ")

if __name__ == '__main__':
    main()
```

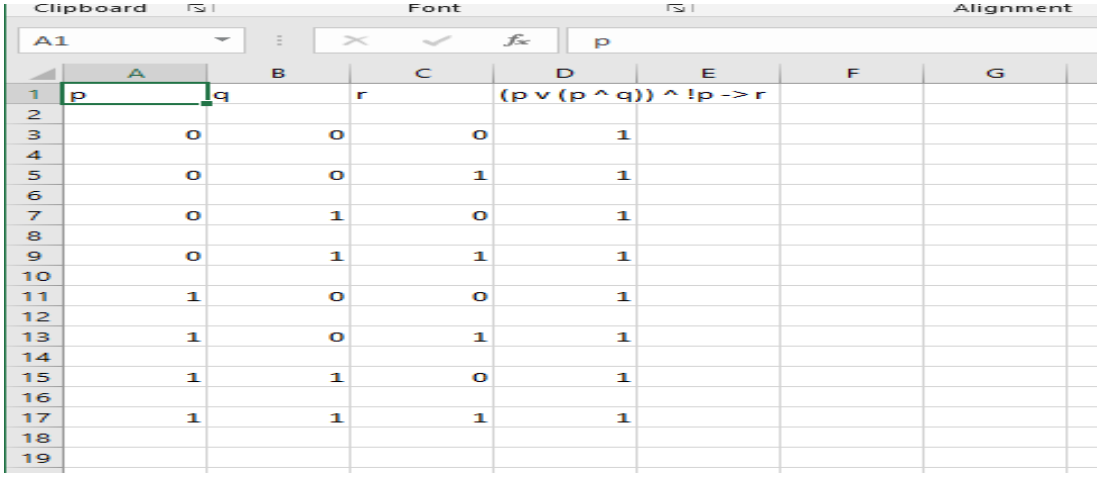
*Et pour finir notre Main function qui lit la list des character du fichier et qui le pass par parameter dans la fonction solve*

# Résultat

## *Résultat console*

```
$ python TruthTableGenerator.py
votre expression =  (p v (p ^ q)) ^ !p -> r
  p   |   q   |   r   |  (p v (p ^ q)) ^ !p -> r
  0   |   0   |   0   |  1
  0   |   0   |   1   |  1
  0   |   1   |   0   |  1
  0   |   1   |   1   |  1
  1   |   0   |   0   |  1
  1   |   0   |   1   |  1
  1   |   1   |   0   |  1
  1   |   1   |   1   |  1
tableau stocker dans result.csv
```

## *Résultat Fichier Excell*




| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | p | q | r | (p v (p ^ q)) ^ !p -> r | | | |
| 2 | | | | | | | |
| 3 | 0 | 0 | 0 | 1 | | | |
| 4 | | | | | | | |
| 5 | 0 | 0 | 1 | 1 | | | |
| 6 | | | | | | | |
| 7 | 0 | 1 | 0 | 1 | | | |
| 8 | | | | | | | |
| 9 | 0 | 1 | 1 | 1 | | | |
| 10 | | | | | | | |
| 11 | 1 | 0 | 0 | 1 | | | |
| 12 | | | | | | | |
| 13 | 1 | 0 | 1 | 1 | | | |
| 14 | | | | | | | |
| 15 | 1 | 1 | 0 | 1 | | | |
| 16 | | | | | | | |
| 17 | 1 | 1 | 1 | 1 | | | |
| 18 | | | | | | | |
| 19 | | | | | | | |

*Vous trouviez le code source dans le lien suivant* [*Link to Project*]
*Consulter ReadMe file*