

TD2 : Logique Booléenne

Exemple de fonction booléenne : $A \Rightarrow B$

On appelle booléens les objets qui sont soit "vrais" (True) soit "faux" (False)

On peut construire avec ces objets des fonctions comme l'implication, le "et" le "ou", etc

```
def implique(l):
    if l[0]:
        return l[1]
    else:
        return True
def et(l):
    if l[0]:
        if len(l)>1: return et(l[1:])
        else: return True
    else: return False
```

1. Expliquez en quoi "implique([A,B])" calcule bien $A \Rightarrow B$
2. Noter que et(l) détermine si tous les éléments de l sont True
3. Écrire de même des fonction "ou" et "equivalent".

Affichage d'un arbre

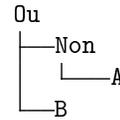
On va considérer dans cette feuille d'exercices certaines fonctions booléennes particulières, qui s'écrivent comme des arbres.

On les stocke sous la forme de listes : par exemple ["Ou", "A", "B"] désigne l'arbre



qui représente l'expression "A ou B"

De même, ["Ou", ["Non", "A"], "B"] désigne l'arbre



qui représente l'expression "(non A) ou B"

```
def affiche_arbre(arb, prefix=""):
    print(prefix+str(arb[0]))
    branches=arb[1:]
    nb_branches=len(branches)
    for i in range(nb_branches):
        branche=branches[i]
        prf=prefix
        for c,d in [(("|", "|"), ("_", " ")), ("L", " ")]:
            prf=str.replace(prf,c,d)
        if i==nb_branches-1:
            prf=prf+"L"
        else:
            prf=prf+"|"
        prf=prf+"-"*len(str(arb[0]))
        affiche_arbre(branche,prf)
```

À noter : cette fonction prend deux arguments : "arb" et "prefix". Mais si le deuxième argument n'est pas donné, alors python considère par défaut qu'il vaut "" (la chaîne de caractères vide)

Par exemple, on peut afficher les arbres suivant

```
arbre_1=["Ou", ["Non", "A"], "B"]
arbre_2=["Et", arbre_1, ["Ou", "A", ["Non", "B"]]]
affiche_arbre(arbre_1)
affiche_arbre(arbre_2)
```

Question optionnelle : comprendre le fonctionnement de cette fonction et expliquer le rôle de l'argument prefix

Valeur booléenne d'un arbre

Étant donné un arbre "arb" et une liste "l" de booléens, la fonction ci-dessous évalue l'arbre lorsque A=l[0], B=l[1], C=l[2], etc

```
def eval_arbre(arb,l):
    if arb[0]=="Ou":return ou([eval_arbre(branche,l) for branche in arb[1:]])
    elif arb[0]=="Et":return et([eval_arbre(branche,l) for branche in arb[1:]])
    elif arb[0]=="Non":return False==eval_arbre(arb[1],l)
    #la boucle ci dessous revient à if arb=="A": return l[0] elif arb=="B": return l[1] elif .....
    for i in range(len(l)):
        if arb==chr(ord("A")+i): return l[i]
    if arb in [True,False]: return arb
    affiche_arbre(arb)
    raise ValueError("Arbre non reconnu")
```

Remarque explicative :

Les caractères comme "A", "B", "C", sont stockés sur ordinateur de la même façon que des nombres. `ord("A")` indique le nombre associé au caractère "A", et `chr(n)` donne le caractère associé au nombre n. En conséquence, `chr(ord("A")+0)` est le caractère "A", alors que `chr(ord("A")+1)` est le caractère "B", etc.

exemple :

```
equiv=["Et",["Ou", ["Non","A"], "B"],["Ou", "A",["Non","B"]]]
print(eval_arbre(equiv,[True,False]))
```

Affichage d'un tableau

Dans la suite, on calculera les tables de vérités de différentes fonctions booléennes.

Pour cela, il est utilise de disposer d'une fonction qui affiche des tableaux :

Les "tableaux" seront stockés comme des "listes de listes" :

la première liste est la première ligne, la deuxième liste est la deuxième ligne, etc

```
def affiche_tableau(tab):
    #Calcul de la largeur maximale de chaque colonne:
    nbline=len(tab) #nombre de lignes du tableau
    nbcol=len(tab[0]) #nombre de colonnes du tableau
    largeur=[]
    for j in range(nbcol):
        largeur=largeur+[max(len(str(tab[i][j])) for i in range(nbline))]
    #affichage ligne par ligne:
    for ligne in tab:
        l="";m=""
        for col in range(nbcol):
            case=str(ligne[col])
            l=l+case+" "*(largeur[col]-len(case))+"| "
            m=m+"-"+largeur[col]+"|"
        print(l)
        print(m)
```

Par exemple, on peut afficher le tableau suivant

```
tbl=[["Nom","Prénom"],["Dalton","Joe"],["Dalton","William"],["Dalton","Jack"]
affiche_tableau(tbl)
```

Liste de cas

On souhaite établir des tables de vérités. Par exemple pour "A<=>B" on aimerait produire la table ci-dessous :

A	B	$A \Leftrightarrow B$
False	False	True
False	True	False
True	False	False
True	True	True

Définir tout d'abord une fonction qui, quand on lui donne le nombre de variables, détermine la liste des cas à considérer dans la table de vérité. Par exemple `liste_cas(2)` doit produire la liste `[[False, False], [False, True], [True, False], [True, True]]` (ou dans un autre ordre)

Table de vérité de fonctions et d'arbres

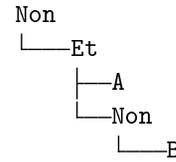
La fonction `"table_verite"` définie ci-dessous affiche la table de vérité d'une liste de fonctions et d'une liste d'arbres, à condition qu'on lui indique le nombre de variables.

```
def table_verite(fonctions,arbres,nombre_de_variables):
    for i in range(len(arbres)):
        print("a"+str(i+1)+" designe l'arbre ci-dessous :")
        affiche_arbre(arbres[i])
    tbl=[[chr(ord("A")+i) for i in range(nombre_de_variables)]+[""]+[f.__name__ for f in fonctions]+["a"+str(i+1) for i in range(len(arbres))]]
    for valeurs in liste_cas(nombre_de_variables):
        tbl=tbl+[valeurs+[""]+[f(valeurs) for f in fonctions]+[eval_arbre(a,valeurs) for a in arbres]]
    affiche_tableau(tbl)
```

exemple :

```
arbre_3=["0u", "A", "B"]
table_verite([equivalent,implique],[arbre_1,arbre_2,arbre_3],2)
```

Déduire que l'arbre `arbre_1` défini ci-avant correspond à la fonction $A \Rightarrow B$, et que l'arbre `arbre_2` correspond à la fonction $A \Leftrightarrow B$. De même, à quelle fonction correspond l'arbre ci-dessous ?



En déduire que deux arbres distincts peuvent correspondre à la même fonction.

Arbre correspondant à une table de vérité fixée

Considérons la fonction `f` dont la table de vérité est donnée ci-dessous :

A	B	f
False	False	False
False	True	True
True	False	False
True	True	False

Elle ne prend la valeur True que si $A=False$ et $B=True$. Elle correspond donc à l'arbre

