Chapitre 2

Manipulation de formules propositionnelles

Dans ce chapitre, nous allons programmer en Python quelques manipulations sur les formules propositionnelles. Le temps prévu pour ce chapitre est de **3 séances** environ, à finir chez soi éventuellement.

Pour ce TP, vous utiliserez le fichier `forumules.py` disponible sur moodle. Pour ce faire, au début de votre fichier il faudra mettre la ligne suivante :

```
from formules import *
```

2.1 Formules logiques et arbres

On s'autorise la conjonction notée `AND`, la disjonction notée `OR`, l'implication notée `IMP`, l'équivalence notée `EQU`, la négation notée `NOT`. Le variables propositionnelles seront notées `X0`, `X1`,... On utilisera aussi `TRUE` et `FALSE`.

Lorsque l'on utilisera une chaîne de caractère pour écrire ou lire une formule (uniquement pour facilité la lecture de l'utilisateur, ce ne sera pas utilisé en machine), on utilisera une notation préfixée. Par exemple la formule

$$\neg(x_1 \wedge x_2) \Rightarrow x_0$$

sera décrite par la chaîne

```
'IMP(NOT(AND(X1,X2)),X0)'
```

Il faudra respecter l'arité des opérateurs, le parenthèsage et ne pas mettre d'espace.

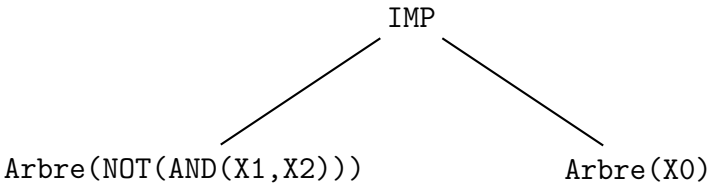
Transformer une formule en arbre binaire se fait alors par induction :

- Si le premier opérateur est `TRUE`, `FALSE` ou une variable, l'arbre n'a qu'un sommet étiqueté par cet opérateur.
- Si le premier opérateur est `NOT`, c'est-à-dire que la formule est du type `NOT(m)`, la racine de l'arbre est étiquetée par `NOT` et l'arbre n'a qu'un fils gauche ; son sous-arbre gauche est l'arbre de `m`.
- Si le premier opérateur est binaire (par exemple `OR`), c'est-à-dire que la formule est du type `OR(m1,m2)`, alors la racine de l'arbre est étiqueté par cet opérateur, a deux fils, le sous arbre gauche est celui de `m1` et le sous arbre droit celui de `m2`.

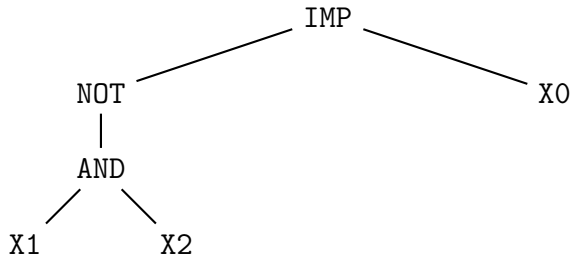
Prenons par exemple la formule

```
'IMP(NOT(AND(X1,X2)),X0)'
```

La formule est du type `IMP(m1,m2)` avec `m1=NOT(AND(X1,X2))` et `m2=X0`. L'arbre sera donc de la forme :



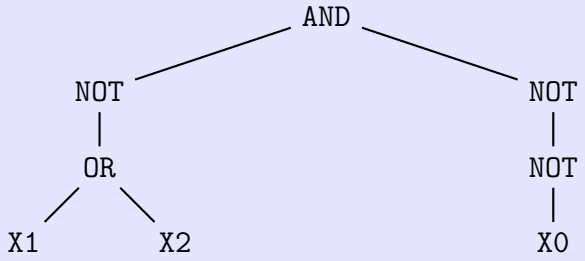
En continuant inductivement sur la formule, on obtient qu'à la fin l'arbre de `IMP(NOT(AND(X1,X2)),X0)` est le suivant :



Q. 1 : Donner en version préfixée la formule $((x_1 \vee x_2) \vee x_3) \Leftrightarrow \neg(x_1 \Rightarrow x_0)$.

Q. 2 : Dessiner l'arbre de cette formule.

Q. 3 : On considère l'arbre ci-dessous



Donner la formule correspondante, sous forme préfixée et sous forme classique.

2.2 Codage machine

Pour coder les formules sous forme d'arbres nous allons utiliser les deux classes `Symbol` et `Node` disponible dans le fichier `formules.py` à inclure (comme décrit en entête de ce chapitre).

Un objet `Symbol` a deux attributs :

- Une arité (`arity`) qui est le nombre d'argument utilisé pour l'opérateur correspondant. La négation a pour arité 1, les variables 0,...
- Un nom (`name`).

La classe est très simple comme on peut le voir ci-dessous et des méthodes spécifiques sont présentes pour les opérateurs utilisés.

```
class Symbol:
    def __init__(self,arity,name):
        self.arity=arity
        self.name=name
    def OR():
        return Symbol(2,"OR")
    def AND():
        return Symbol(2,"AND")
    def IMP():
```

```
        return Symbol(2,"IMP")
    def EQU():
        return Symbol(2,"EQU")
    def T():
        return Symbol(0,"TRUE")
    def F():
        return Symbol(0,"FALSE")
    def NOT():
        return Symbol(1,"NOT")
    def VAR(i):
        return Symbol(0,"X"+str(i))
```

Voici un exemple :

```
from formules import *

mySymb=Symbol.OR()
print(mySymb.name)
print(mySymb.arity)
myVar=Symbol.VAR(5)
print(myVar.name)
print(myVar.arity)
```

Va retourner

```
OR
2
X5
0
```

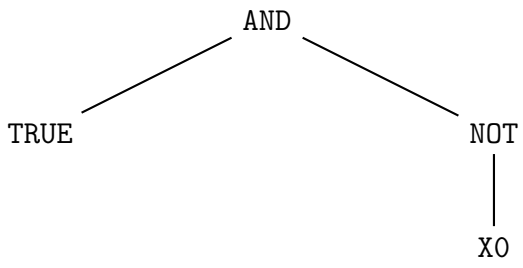
La classe `Node` va permettre de construire les arbres des formules :

```
class Node:
    def __init__(self,S):
        self.left=None
        self.right=None
        self.value=S
```

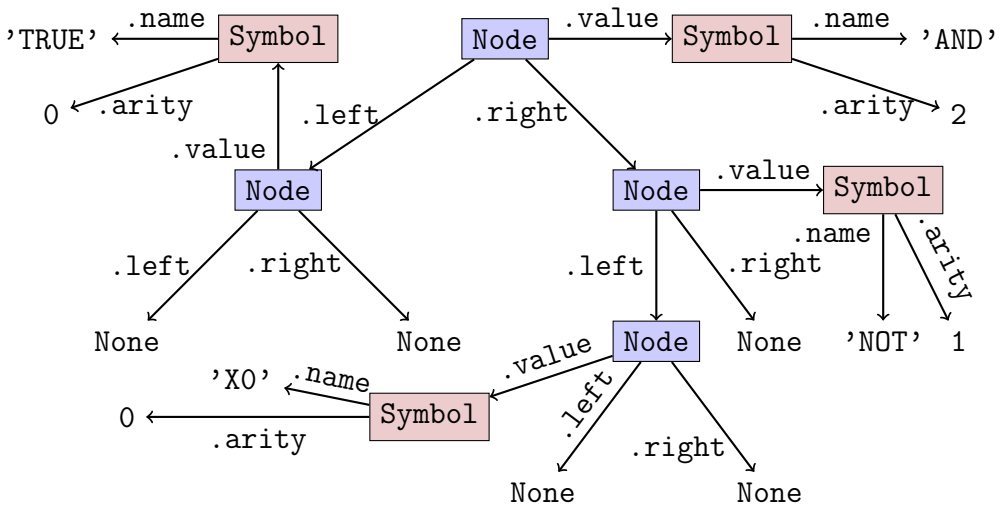
Chaque objet `Node` correspondra à un noeud de l'arbre. Il a trois attributs :

- `value`, qui est un symbole, celui qui sera dans le noeud de l'arbre.
- `left` et `right` qui sont aussi des objets `Node` contenant les sous-arbres gauche et droit. Pour les symboles d'arité 0, ils sont à `None` (vide), pour `NOT` seul le sous-arbre gauche est utilisé.

Considérons par exemple la formule `AND(TRUE,NOT(X0))` dont l'arbre est



sera codé en mémoire par :



Cela peut paraître, a priori complexe : il y a une structuration (en bleue) qui correspond à la forme de l'arbre de la formule. Ensuite, à chaque sommet bleu on associe une valeur (`.value`) qui est un objet `Symbol`, qui a un attribut `.name` et un attribut `.arity`.

Si l'on veut construire explicitement cet arbre, on peut utiliser le code suivant :

```
exampleForm=Node(Symbol.AND()) #creation de la racine
print(exampleForm)
exampleForm.left=Node(Symbol.T()) # creation du fils gauche
exampleForm.right=Node(Symbol.NOT()) # creation du fils droit
print(exampleForm)
exampleForm.right.left=Node(Symbol.VAR(0)) # dernier noeud
print(exampleForm)
```

Exécuter le code ci-dessus. Écrire un code similaire pour la formule `OR(AND(TRUE,FALSE),NOT(NOT(X1)))`.

Comme cela peut être un peu lourd à la main, la classe `Node` dispose d'une méthode `.fromString` qui à partir d'une chaîne de caractères d'une formule construit l'objet correspondant en mémoire.

La structure arborescente est très efficace pour une utilisation machine. Voici, par exemple, un code permettant de calculer le nombre de symboles utilisés dans une formule (nombre de sommets de l'arbre correspondant).

```
def nombreSym(formule):
    if formule.value.arity==0:
        return 1
    if formule.value.arity==1:
        return 1+nombreSym(formule.left)
    return 1+nombreSym(formule.left)+nombreSym(formule.right)
```

Q. 1 : Écrire une fonction `nbNot(formule)` qui étant donné une formule, retourne le nombre de `NOT` qu'elle contient.

Q. 2 : Écrire une fonction `listeVariables(formule)` qui étant donnée une formule, retourne la liste des variables qu'elles contient (sous forme d'un tableau d'entier).

Q. 3 : Écrire une fonction `conjonction(formule1,formule2)` qui étant données deux formules, retourne la conjonction de ces deux formules, sans recopier les deux formules (c'est-à-dire qu'après appel de la fonction, si l'on modifie l'une des deux formules, cela modifie aussi la conjonction calculée).

2.3 Algorithmes pour la logique propositionnelle

Q. 1 : Écrire une fonction `evalFormuleClose(formule)` qui étant donnée une formule close, retourne un booléen indiquant si elle est vraie ou fausse. Celles et ceux qui sont en avance sur les TP peuvent pour cette question réfléchir à des optimisations évitant de parcourir systématiquement toute la formule).

Q. 2 : Écrire une fonction `evalFormule(formule,instanciation)` qui étant donnée une formule, une instanciation (donnée par un dictionnaire qui à chaque numéro de variable donne le booléen correspondant à l'instanciation), indique si cette interprétation est vraie ou fausse.

Q. 3 : Encire une fonction `tableDeVerite(formule)` qui calcule la table de vérité d'une formule (à ne tester que sur de petites formules).

Q. 4 : Écrire une fonction `supprimeIMPracine(formule)` qui supprime le `IMP` à la racine d'une formule (s'il y en a un) en utilisant la règle de réécriture :

$$\texttt{IMP(A,B)} \rightarrow \texttt{OR(NOT(A),B)}$$

Q. 5 : Écrire une fonction `supprimeIMP(formule)` qui supprime tous les `IMP` d'une formule en utilisant la règle de réécriture :

$$\texttt{IMP(A,B)} \rightarrow \texttt{OR(NOT(A),B)}$$

Q. 6 : Écrire une fonction `supprimeEQUracine(formule)` qui supprime l'`EQU` à la racine d'une formule (s'il y en a un) en utilisant la règle de réécriture :

$$\texttt{EQU(A,B)} \rightarrow \texttt{AND(OR(NOT(A),B),OR(NOT(B),A))}$$

Q. 7 : Écrire une fonction `supprimeEQU(formule)` qui supprime tous les `EQU` d'une formule en utilisant la règle de réécriture :

$$\texttt{EQU(A,B)} \rightarrow \texttt{AND(OR(NOT(A),B),OR(NOT(B),A))}$$

Q. 8 : Écrire une fonction qui transforme une formule en une formule équivalente n'ayant que des `OR`, `AND`, `TRUE`, `FALSE`, `NOT`, des variables, et telle que tous les `NOT` aient pour fils une variable (la démarche a été vue en cours).