

## Projet Info LDD2 – TD 9

**Objectifs du TD :** Synthèse de circuit via une formule propositionnelle.


L'objectif de ce TD va être de construire un circuit booléen à partir d'une formule propositionnelle donnée sous la forme d'une chaîne de caractères. Pour se simplifier la vie, on considérera que la formule est donnée sous forme complètement parenthésée, et en notation infixe. On aura par exemple " $((x0) \& ((x1) \& (x2))) \mid ((x1) \& (\sim(x2)))$ ". Cela permet d'une part de ne pas avoir à mettre en place les règles de priorité de calcul, et cela permet d'autre part de simplifier le *parsing* de la formule.

Le premier objectif va être de créer l'arbre de la formule, ce qu'on va représenter par un `bool_circ` avec les noeuds qui contiennent comme `label` les connecteurs logiques, ou bien les noms de variable (ces noms de variables n'étant pas des labels valables, on les retirera dans la suite). L'algorithme pour faire cela est donné ci-après.

---

**Algorithm 1** Formule propositionnelle vers arbre

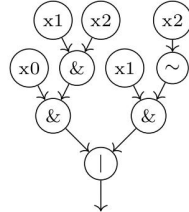
---

```
function PARSE_PARENTHESSES(s)           ▷ s est une chaîne de caractères
    g ←  (i.e. le circuit booléen à un noeud relié à une sortie)
    current_node ← l'id du noeud du dessus
    s2 ← ''
    for all char in s do
        if char = '(' then
            rajouter s2 au label de current_node
            créer un parent à current_node et faire en sorte que ce parent soit
                                                    désormais current_node
            s2 ← ''
        else if char = ')' then
            rajouter s2 au label de current_node
            changer current_node pour qu'il prenne l'id de son fils
            s2 ← ''
        else
            ajouter char à la fin de s2
        end if
    end for
    return g
end function
```

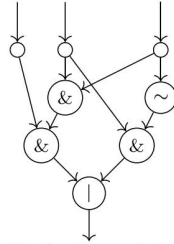
---

┌ **Exercice 1 :** Implémenter cette méthode de `bool_circ`. ─

Voici ce qu'on devrait obtenir à partir de la formule précédente :



Ceci n'est techniquement pas un circuit booléen, mais on va remédier à cela dans la suite. Pour rappel, voici ce que l'on cherche à obtenir :

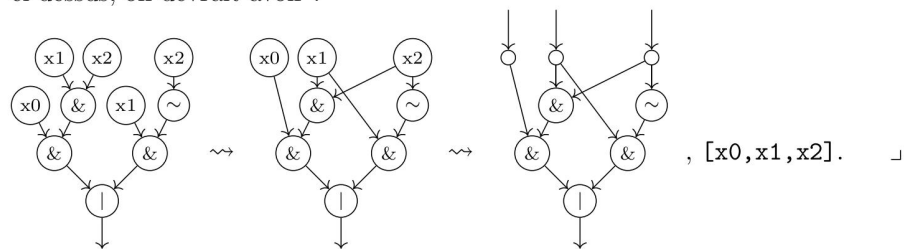


Vous remarquerez qu'on a un opérateur de copie à une seule sortie. Ça n'est pas gênant, cela correspond simplement à de l'information que l'on a copié ... 0 fois.

Il s'agit maintenant de regrouper les différentes occurrences des variables en un seul noeud de copie, relié à une entrée. Pour ce faire, un outil utile est la fusion de deux noeuds.

▮ **Exercice 2 :** Implémenter une méthode d'`open_digraph` qui fusionne deux noeuds dont les id sont donnés en paramètres. On peut choisir un des deux labels comme celui par défaut, ou bien demander à l'utilisateur le label à donner à la fusion des deux noeuds. ▮

▮ **Exercice 3 :** Modifier l'algorithme de l'exo 1 pour regrouper les variables et obtenir un vrai circuit booléen. Les noms de variables doivent être supprimés pour avoir un circuit bien formé. Renvoyer en plus du circuit la liste des noms de variable dont l'ordre correspond à celui des inputs : le  $i$ ème input du circuit doit correspondre à la  $i$ ème variable dans cette liste. Par exemple, avec le circuit ci-dessus, on devrait avoir :



Dans un circuit booléen, on a droit à avoir plusieurs sorties, chacune correspondant à une formule booléenne.

▮ **Exercice 4 :** Modifier encore l'algorithme pour qu'il prenne cette fois `*args` une séquence de chaînes de caractères en paramètre. Chaque chaîne de caractères est supposée être une formule propositionnelle comme précédemment. On doit obtenir à la fin un seul circuit booléen qui implémente la séquence (i.e. chaque

Astuce : on ne s'intéresse pas ici à optimiser le circuit, on peut simplement construire les arbres des différentes formules côte à côte, puis faire l'opération qui relie les variables. »

peut donner :

「 **Exercice 5** (Bonus) : Modifier une dernière fois l'algorithme pour régler les problèmes (s'il y en a) lorsqu'on ôte les parenthèses inutiles comme dans l'exemple précédent. 」