Formules Logiques

Marc Lorenzi - 20 avril 2018

In [1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import random, sys
```

1. Syntaxe des formules

1.1 Notion de formule

On se donne un ensemble $\mathcal{V}$ dont les éléments sont appelés les **variables propositionnelles**. La nature exacte de $\mathcal{V}$ n'a pas d'importance. De façon informelle, lorsque nous aurons besoin de variables nous les noterons x, y, z, a, b, a_0 , etc.

On pose également $\mathcal{S} = \{-, +, ., \rightarrow, \leftrightarrow\ (,)\}$. Nous appellerons formule toute suite de symboles (les informaticiens disent "chaîne de caactères", les théoriciens disent "mot") de $\mathcal{V} \cup \mathcal{S}$ vérifiant les propriétés suivantes :

- Les éléments de $\mathcal{V}$ sont des formules.
- Si f est une formule, $-f$ est une formule.
- Si f_1 et f_2 sont des formules, $(f_1 . f_2)$ est une formule.
- Si f_1 et f_2 sont des formules, $(f_1 + f_2)$ est une formule.
- Si f_1 et f_2 sont des formules, $(f_1 \rightarrow f_2)$ est une formule.
- Si f_1 et f_2 sont des formules, $(f_1 \leftrightarrow f_2)$ est une formule.

"$-$" est le **connecteur unaire** (il symbolise la négation). "$+$", "$.$", "$\rightarrow$" et "$\leftrightarrow$" sont les **connecteurs binaires**. Le point symbolise le "et", le $+$ représente le "ou", et les flèches $\rightarrow$ et $\leftrightarrow$ représentent l'implication et l'équivalence. Le choix des symboles de connecteurs est guidé par le fait qu'ils sont faciles à entrer au clavier.

On adopte également des conventions de priorité qui permettent de ne pas écrire certaines parenthèses : les connecteurs, du plus prioritaire au moins prioritaire, sont $-, ., +, \rightarrow, \leftrightarrow$. Ainsi, par exemple, $(x \rightarrow y) + (y \rightarrow z) \leftrightarrow (x \rightarrow z)$ est une façon abrégée d'écrire la formule $(((x \rightarrow y) + (y \rightarrow z)) \leftrightarrow (x \rightarrow z))$.

1.2 Représenter les formules en Python

Nous allons dans ce notebook écrire des fonctions qui prennent des formules en paramètre, qui calculent des formules, qui les combient entre-elles ... Il s'agit d'avoir une représentation des formules dans notre langage préféré qui nous permette de les manipuler efficacement. Cette formule est elle un "et" ? Une implication ? Quelles sont ses variables ? etc.

On peut distinguer trois sortes de formules.

- Les variables. Nous les représenterons par un couple `('var', x)` où x est une chaîne de caractères, qui est le nom proprement dit de la variable.
- La négation d'une formule. Nous représenterons une négation par le couple `('not', f1)` où f_1 est elle même la représentation d'une formule.
- Les formules "binaires". Nous représenterons une telle formule par un triplet `(symb, f1, f2)` où `symb` peut prendre les valeurs `and`, `or`, `imp` et `eqv` (et, ou, implication et équivalence).

Voici un exemple, la représentation de la formule $(x \to y) + (y \to z) \leftrightarrow (x \to z)$.

In [2]:
```
exemple = ('eqv',
     ('and', ('imp', ('var', 'x'), ('var', 'y')),
             ('imp', ('var', 'y'), ('var', 'z'))
     ),
     ('imp', ('var', 'x'), ('var', 'z')))
```

On voit tout de suite apparaître deux problèmes pratiques :

1. C'est impossible à lire.
2. C'est impossible à écrire.

Nous allons petit à petit régler ces deux problèmes.

1.3 Affichage des formules

Résolvons déjà le premier problème : l'affichage d'une formule. Nous allons écrire une fonction qui, étant donnée une formule f, renvoie une représentation de f sous forme d'une chaîne de caractères lisible par un être humain (enfin par un logicien).

Voici deux fonctions évidentes. Pour une formule f de type `('not', g)`, `left(f)` renvoie g. Pour une formule binaire, `left` et `right` renvoient les constituants de la formule.

In [3]:
```
def left(f): return f[1]
def right(f): return f[2]
```

In [4]:
```
left(exemple)
```

Out[4]:
```
('and',
 ('imp', ('var', 'x'), ('var', 'y')),
 ('imp', ('var', 'y'), ('var', 'z')))
```

In [5]:
```
right(exemple)
```

Out[5]:
```
('imp', ('var', 'x'), ('var', 'z'))
```

Le dictionnaire `symbol` contient la représentation symbolique de chacun des connecteurs.

In [6]:
```
symbol = {'not': '-', 'and': '.', 'or': '+', 'imp': '->', 'eqv': '<->'
```

In [7]:
```
symbol['eqv']
```

Out[7]:
```
'<->'
```

Le dictionnaire `prio` contient les priorités de tous les connecteurs. On a également assigné une priorité maximale aux variables.

In [8]:
```
prio = {'var': 15, 'not': 10, 'and': 8, 'or': 6, 'imp': 4, 'eqv': 2}
```

La fonction `paren` prend en paramètres une chaîne de caractères s et deux formules. Selon la priorité relative de ces formules, elle renvoie la chaîne s entre parenthèses ou pas.

In [9]:
```
def paren(s, f1, f):
    if prio[f1[0]] <= prio[f[0]]: return '(' + s + ')'
    else: return s
```

Et voici enfin la fonction `to_string` . Elle prend une formule f en paramètre et renvoie une chaîne qui est une forme "humainement lisible" de la formule. Le code est simple, étudiez-le. En particulier, comprenez à quoi sert l'appel à `paren` .

In [10]:
```
def to_string(f):
    if f[0] == 'var': return f[1]
    elif f[0] == 'not':
        f1 = left(f)
        s1 = to_string(f1)
        return '-' + paren(s1, f1, f)
    else:
        smb = symbol[f[0]]
        f1 = left(f)
        f2 = right(f)
        s1 = to_string(f1)
        s2 = to_string(f2)
        return paren(s1, f1, f) + smb + paren(s2, f2, f)
```

In [11]:
```
to_string(exemple)
```

Out[11]:
```
'(x->y).(y->z)<->x->z'
```

In [12]:
```
exemple
```

Out[12]:
```
('eqv',
 ('and',
  ('imp', ('var', 'x'), ('var', 'y')),
  ('imp', ('var', 'y'), ('var', 'z'))),
 ('imp', ('var', 'x'), ('var', 'z')))
```

1.4 Hauteur d'une formule

La hauteur $h(f)$ d'une formule f est définie comme suit :

- La hauteur d'une variable est 0.
- Si $f = -f_1$, alors $h(f) = 1 + h(f_1)$.
- Si $f = f_1 \alpha f_2$, où α est un connecteur binaire alors $h(f) = 1 + \max(h(f_1), h(f_2))$.

Pourquoi appeler cela "hauteur" ? Lorsque nous verrons qu'à chaque formule on peut associer un arbre, on comprendra mieux. Ou alors, si on a déjà lu le notebook sur les arbres, on a déjà compris.

In [13]:
```
def hauteur(f):
    if f[0] == 'var': return 0
    elif f[0] == 'not' : return 1 + hauteur(left(f))
    else:
        h1 = hauteur(left(f))
        h2 = hauteur(right(f))
        return 1 + max([h1, h2])
```

In [14]:
```
hauteur(exemple)
```

Out[14]: 3

1.5 Formules "aléatoires"

Pour tester les fonctions que nous allons écrire, il peut être intéressant de pouvoir fabriquer des formules un peu n'importe comment et très très compliquées. La fonction ci-dessous renvoie une formule "aléatoire" de hauteur h dont les variables sont $a_0, a_1, \ldots, a_{n-1}$.

In [15]:
```
def formule_aleatoire(h, n):
    if h == 0:
        p = random.randint(0, n - 1)
        return ('var', 'a' + str(p))
    else:
        f1 = formule_aleatoire(h - 1, n)
        p = random.randint(0, 4)
        if p == 0: return ('not', f1)
        else:
            h1 = random.randint(0, h - 1)
            f2 = formule_aleatoire(h1, n)
            b = random.randint(0, 1)
            if b == 1: f1, f2 = f2, f1
            if p == 1: return ('and', f1, f2)
            elif p == 2: return ('or', f1, f2)
            elif p == 3: return ('imp', f1, f2)
            else: return ('eqv', f1, f2)
```

In [16]:
```
f = formule_aleatoire(10, 5)
to_string(f)
```

Out[16]:
```
'-(a0<->(a2.a2).(a2.a4)+(a3+(a3<->a1))->(a2->(a4->a2)<->-a0+a4)+(-(-
((a2+a3)+a0).(a1<->(a4.a2).(a0.a2)))+((-a3+a3)+-(a0<->a0))))'
```

In [17]:
```
f
```

Out[17]:
```
('not',
 ('eqv',
  ('var', 'a0'),
  ('imp',
   ('or',
    ('and',
     ('and', ('var', 'a2'), ('var', 'a2')),
     ('and', ('var', 'a2'), ('var', 'a4'))),
    ('or', ('var', 'a3'), ('eqv', ('var', 'a3'), ('var', 'a1')))),
   ('or',
    ('eqv',
     ('imp', ('var', 'a2'), ('imp', ('var', 'a4'), ('var', 'a2'))),
     ('or', ('not', ('var', 'a0')), ('var', 'a4'))),
    ('or',
     ('not',
      ('and',
       ('not', ('or', ('or', ('var', 'a2'), ('var', 'a3')), ('var',
'a0'))),
       ('eqv',
        ('var', 'a1'),
        ('and',
         ('and', ('var', 'a4'), ('var', 'a2')),
         ('and', ('var', 'a0'), ('var', 'a2')))))),
     ('or',
      ('or', ('not', ('var', 'a3')), ('var', 'a3')),
      ('not', ('eqv', ('var', 'a0'), ('var', 'a0')))))))))
```

In [18]:
```
hauteur(f)
```

Out[18]:
```
10
```

Ben oui c'est normal.

1.6 Arbre associé à une formule

Je vais ici supposer que vous avez jeté un coup d'oeil au notebook sur les arbres. Je ne reprendrai pas la définition de racine, de fils, de noeud, de feuille, etc. Si ce qui suit est d'une obscurité totale pour vous, sautez les explications et foncez à l'exemple !

Eh oui, une formule, par exemple $f_1 + f_2$ peut être vue comme un arbre dont la racine est $+$ et les deux fils sont les arbres associés aux formules f_1 et f_2. Selon le type de la formule f, l'arbre qui la représente a différentes formes :

- si f est une variable x, l'arbre a juste un noeud, d'étiquette x.
- Si $f = -g$, l'arbre a une racine étiquetée par $\sim$ et un seul fils qui est l'arbre qui représente g.
- Si $f = f_1.f_2$, l'arbre a une racine étiquetée par . et deux fils qui sont les arbre qui représentent f_1 et f_2. Et de même pour les autres connecteurs binaires.

La fonction `draw(f)` ci-dessous dessine l'arbre qui représente la formule `f`. Elle utilise une fonction auxiliaire `draw_aux`. Dans le notebook sur les arbres apparaissent des fonctions quasiment identiques.

In [19]:
```
def draw_aux(f, rect, dy):
    x1, x2, y1, y2 = rect
    xm = (x1 + x2) // 2
    if f[0] == 'var': noeud = f[1]
    else: noeud = symbol[f[0]]
    plt.text(xm + 3, y2, noeud, fontsize=12, horizontalalignment='left
    if f[0] == 'var': return
    if f[0] == 'not':
        draw_aux(left(f), (x1, x2, y1, y2 - dy), dy)
        a, b = ((xm, xm), (y2, y2 - dy))
        plt.plot(a, b, 'k', marker='o')
    else:
        draw_aux(left(f), (x1, xm, y1, y2 - dy), dy)
        draw_aux(right(f), (xm, x2, y1, y2 - dy), dy)
        a, b = ((xm, (x1 + xm) // 2), (y2, y2 - dy))
        plt.plot(a, b, 'k', marker='o')
        c, d = ((xm, (x2 + xm) // 2), (y2, y2 - dy))
        plt.plot(c, d, 'k', marker='o')
```

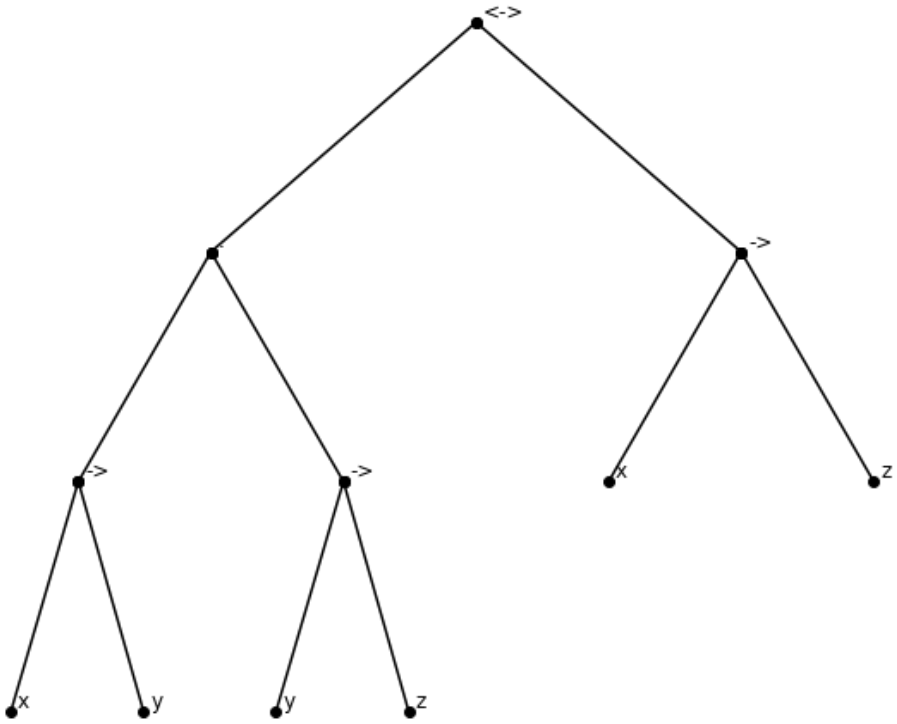
In [20]:
```
def draw(f):
    d = 512
    pad = 20
    dy = (d - 2 * pad) / (hauteur(f))
    draw_aux(f, (pad, d - pad, pad, d - pad), dy)
    plt.axis([0, d, 0, d])
    plt.axis('off')
    plt.show()
```

La ligne suivante, c'est pour que nos arbres soient affichés assez gros à l'écran.

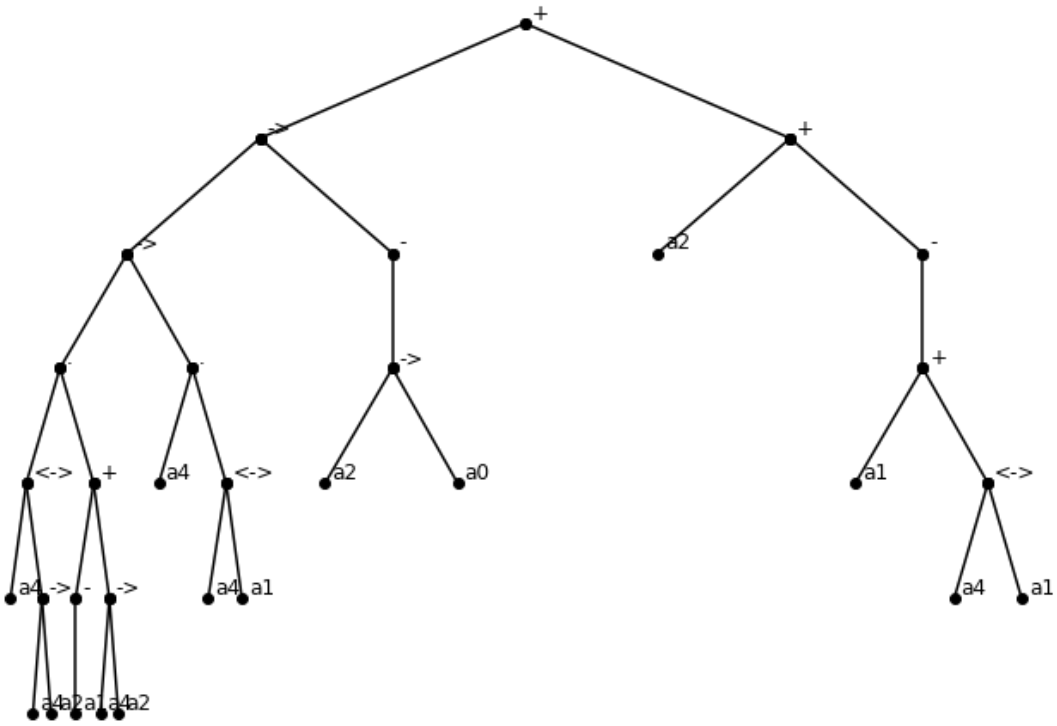
In [21]:
```
plt.rcParams['figure.figsize'] = (12, 8)
```

Testons `draw` ...

In [22]:
```
draw(exemple)
```



In [23]:
```
f = formule_aleatoire(6, 5)
draw(f)
```



1.7 Les variables d'une formule

Quelles sont les variables qui interviennent dans une formule ? La fonction ci-dessous résout le problème.

In [24]:
```
def variables(f):
    if f[0] == 'var': return set([f[1]])
    elif f[0] == 'not': return variables(f[1])
    else: return variables(f[1]).union(variables(f[2]))
```

In [25]:
```
variables(exemple)
```

Out[25]:
```
{'x', 'y', 'z'}
```

In [26]:
```
variables(formule_aleatoire(8, 6))
```

Out[26]:
```
{'a0', 'a1', 'a2', 'a3', 'a4', 'a5'}
```

2. Un analyseur syntaxique

Ce paragraphe n'est pas facile. Il peut être sauté sans honte en première lecture.

L'ensemble des formules logiques est un langage, c'est à dire un ensemble de mots, dont les lettres appartiennent à l'alphabet $\mathcal{A} = \mathcal{V} \cup \mathcal{S}$ que nous avons défini au début de ce notebook. Étant donné un mot sur l'alphabet $\mathcal{A}$ qui représente une formule valide, nous voudrions décrire un algorithme qui permet de l'analyser pour en déduire la représentation en Python de la formule. Histoire de compliquer un peu les choses, on autorise un parenthésage "minimal".

La tâche paraît ardue.

2.1 Premier jet

Appelons irréductible toute formule qui ne contient que des parenthèses, des "not", et une variable. Une telle formule peut être de trois sortes :

- x, où x est une variable
- (f) où f est une formule irréductible
- $-f$ où f est irréductible.

On peut résumer cela par la ligne

```
I ::= -I  | V | (I)
```

La barre verticale signifie "ou bien", V et I signifient respectivement "variable" et "formule irréductible"

La fonction `parseI` ci-dessous prend une chaîne de caractères s en paramètre. Elle renvoie un couple (f, r) où f est le formule irréductible représentée par le plus petit **préfixe** possible de s, et r est la partie de s qui n'a servi à rien (c'est un **suffixe** de s).

Remarque : un préfixe d'une chaîne de caractères s est une chaîne constituée des premières lettres de s. Par exemple, les préfixes de 'bonjour' sont '', 'b', 'bo'n 'bon', ..., 'bonjour'. Un préfixe de s différent s est dit **strict**. On définit bien sûr de même la notion de **suffixe**.

In [27]:
```
def parseI(s):
    if s[0] == '(':
        f, reste = parseI(s[1:])
        return (f, reste[1:])
    elif s[0] == '-':
        f, reste = parseI(s[1:])
        return (('not', f), reste)
    else:
        return (('var', s[0]), s[1:])
```

Par exemple, prenons $s = --(-((x)))(y \to z)$. La partie intéressante du début de s est $--(-((x)))$. Le reste est $(y \to z)$.

In [28]:
```
parseI('--(-((x)))(y->z)')
```

Out[28]:
```
(('not', ('not', ('not', ('var', 'x')))), '(y->z)')
```

2.2 On complique un peu

Tâchons maintenant d'analyser des formules contenant des "et", des "not", des variables et des parenthèses. Une formule de cette sorte est essentiellement

- une formule irréductible, ou bien
- une formule irréductible "et" une autre formule de cette sorte

Notons `E4` une telle formule : `E4 ::= I . E4 | I`

Oui, d'accord, à condition de redéfinir `I` en : `I ::= -I | V | (E4)` . Eh oui, une expression de type `E4` avec une paire de parenthèses englobantes doit être considérée comme irréductible ! Le code Python devient donc :

In [29]:
```
def parse4(s):
    f, reste = parseI(s)
    if len(reste) >= 1 and reste[0] == '.':
        f1, reste1 = parse4(reste[1:])
        return (('and', f, f1), reste1)
    else:
        return (f, reste)
```

In [30]:
```
def parseI(s):
    if s[0] == '(':
        f, reste = parse4(s[1:])
        return (f, reste[1:])
    elif s[0] == '-':
        f, reste = parseI(s[1:])
        return (('not', f), reste)
    else:
        return (('var', s[0]), s[1:])
```

Testons ...

In [31]:
```
parse4('a.(a.-b)c->d')
```

Out[31]:
```
(('and', ('var', 'a'), ('and', ('var', 'a'), ('not', ('var', 'b')))),
 'c->d')
```

Le premier préfixe de notre exemple qui est de type `E4` est correctement analysé, et tout ce qui suit est renvoyé sans y toucher.

2.3 On rajoute les +

Attaquons nous à l'analyse des formules contenant des "ou", des "et", des "not", des variables et des parenthèses. Une formule de cette sorte est essentiellement

- une formule du type `E4` , ou bien
- une formule du type `E4` "ou" une autre formule de cette sorte

Notons `E3` une telle formule : `E3 ::= E4 . E3 | E4`

Pourquoi cela fonctionne-t-il ? parce que le "et" est prioritaire par rapport au "ou" !

Le code Python devient donc :

In [32]:
```
def parse3(s):
    f, reste = parse4(s)
    if len(reste) >= 1 and reste[0] == '+':
        f1, reste1 = parse3(reste[1:])
        return (('or', f, f1), reste1)
    else:
        return (f, reste)
```

In [33]:
```
def parse4(s):
    f, reste = parseI(s)
    if len(reste) >= 1 and reste[0] == '.':
        f1, reste1 = parse4(reste[1:])
        return (('and', f, f1), reste1)
    else:
        return (f, reste)
```

In [34]:
```
def parseI(s):
    if s[0] == '(':
        f, reste = parse4(s[1:])
        return (f, reste[1:])
    elif s[0] == '-':
        f, reste = parseI(s[1:])
        return (('not', f), reste)
    else:
        return (('var', s[0]), s[1:])
```

Testons ...

In [35]:
```
parse3('a.b+-(c.d)+e->f')
```

Out[35]:
```
(('or',
  ('and', ('var', 'a'), ('var', 'b')),
  ('or', ('not', ('and', ('var', 'c'), ('var', 'd'))), ('var', 'e'))
),
 '->f')
```

Le premier préfixe de notre exemple qui est de type `E3` est correctement analysé, et tout ce qui suit, c'est à dire $\rightarrow f$, est renvoyé sans y toucher.

2.4 On rajoute implications et équivalences

Rajouter les derniers connecteurs ne pose pas de problème supplémentaires. Il faut juste prendre garde à leur priorité : les moins prioritaires d'abord. Appelons `E1` le type des formules qui sont des équivalences et `E2` le type des formules qui sont des implications. Voici enfin la **grammaire** complète des formules :

```
E1 ::= E2 <-> E1 | E2

E2 ::= E3 ->  E2 | E3

E3 ::= E4  +  E3 | E4

E4 ::= I   .  E4 | I

I ::= -I  | V | (E1)
```

Je vous suggère d'écrire quelques formules et de réfléchir à la question avant de poursuivre ...

Notre analyseur de formules est donc composé de 5 fonctions, une pour chaque ligne de la grammaire. Chacune des fonctions appelle la fonction de numéro "un de plus", sauf la dernière qui peut se permettre d'appeler la première. Oui, nous avons là 5 fonctions mutuellement récursives.

Chacune de ces fonctions prend une chaine s de caractères. Elle analyse le début de s jusqu'à trouver une formule bien formée pour la fonction en question. Puis la fonction renvoie un couple formé de la formule trouvée et du reste non analysé de s.

In [36]:
```
def parse1(s):
    f, reste = parse2(s)
    if len(reste) >= 3 and reste[0:3] == '<->':
        f1, reste1 = parse1(reste[3:])
        return (('eqv', f, f1), reste1)
    else:
        return (f, reste)
```

In [37]:
```
def parse2(s):
    f, reste = parse3(s)
    if len(reste) >= 2 and reste[0:2] == '->':
        f1, reste1 = parse2(reste[2:])
        return (('imp', f, f1), reste1)
    else:
        return (f, reste)
```

In [38]:
```
def parse3(s):
    f, reste = parse4(s)
    if len(reste) >= 1 and reste[0] == '+':
        f1, reste1 = parse3(reste[1:])
        return (('or', f, f1), reste1)
    else:
        return (f, reste)
```

In [39]:
```
def parse4(s):
    f, reste = parseI(s)
    if len(reste) >= 1 and reste[0] == '.':
        f1, reste1 = parse4(reste[1:])
        return (('and', f, f1), reste1)
    else:
        return (f, reste)
```

In [40]:
```
def parseI(s):
    if s[0] == '(':
        f, reste = parse1(s[1:])
        return (f, reste[1:])
    elif s[0] == '-':
        f, reste = parseI(s[1:])
        return (('not', f), reste)
    else:
        return (('var', s[0]), s[1:])
```

Pour analyser la chaîne s, on appelle `parse1`, qui renvoie un couple (f, r) et on laisse tomber r ... qui devrait être la chaîne vide. Disons-le clairement, cet analyseur n'est pas bien solide :

- il n'accepte que des variables propositionnelles de 1 caractère
- il ne supporte pas les espaces
- il ne fait aucun effort pour détecter les erreurs de syntaxe

Mais il suffira pour ce que nous voulons en faire : taper rapidement des exemples.

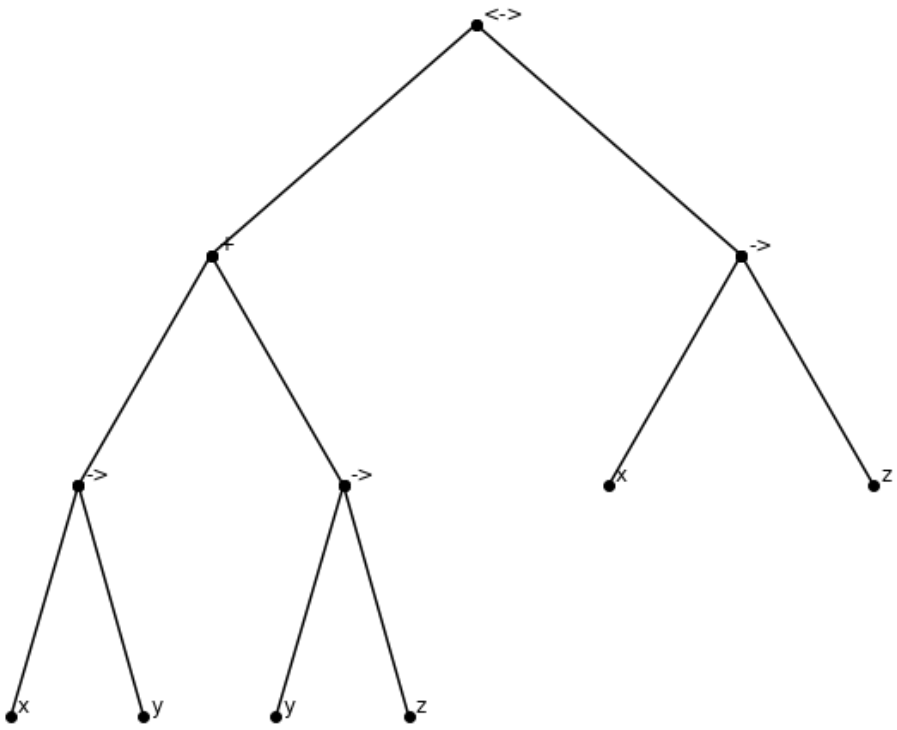
In [41]:
```
def parse(s):
    f, reste = parse1(s)
    return f
```

Testons !

In [42]:
```
f = parse('(x->y)+(y->z)<->x->z')
f
```

Out[42]:
```
('eqv',
 ('or',
  ('imp', ('var', 'x'), ('var', 'y')),
  ('imp', ('var', 'y'), ('var', 'z'))),
 ('imp', ('var', 'x'), ('var', 'z')))
```

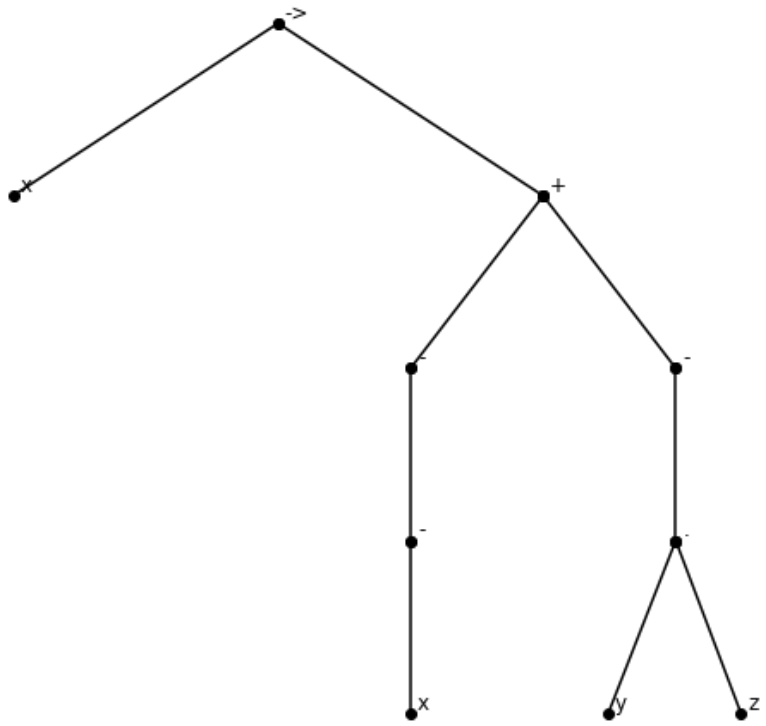
In [43]:
```
draw(f)
```



In [44]:
```
f = parse('x->--x+-(y.z)')
f
```

Out[44]:
```
('imp',
 ('var', 'x'),
 ('or',
  ('not', ('not', ('var', 'x'))),
  ('not', ('and', ('var', 'y'), ('var', 'z')))))
```

In [45]:
```
draw(f)
```



Vous pouvez maintenant très facilement utiliser avec Python vos formules logiques préférées.

3. Disséquer les formules

Juste pour voir ... Voici quelques illustrations de ce que nous pouvons faire avec notre représentation des formules.

3.1 Éliminer les implications et les équivalences

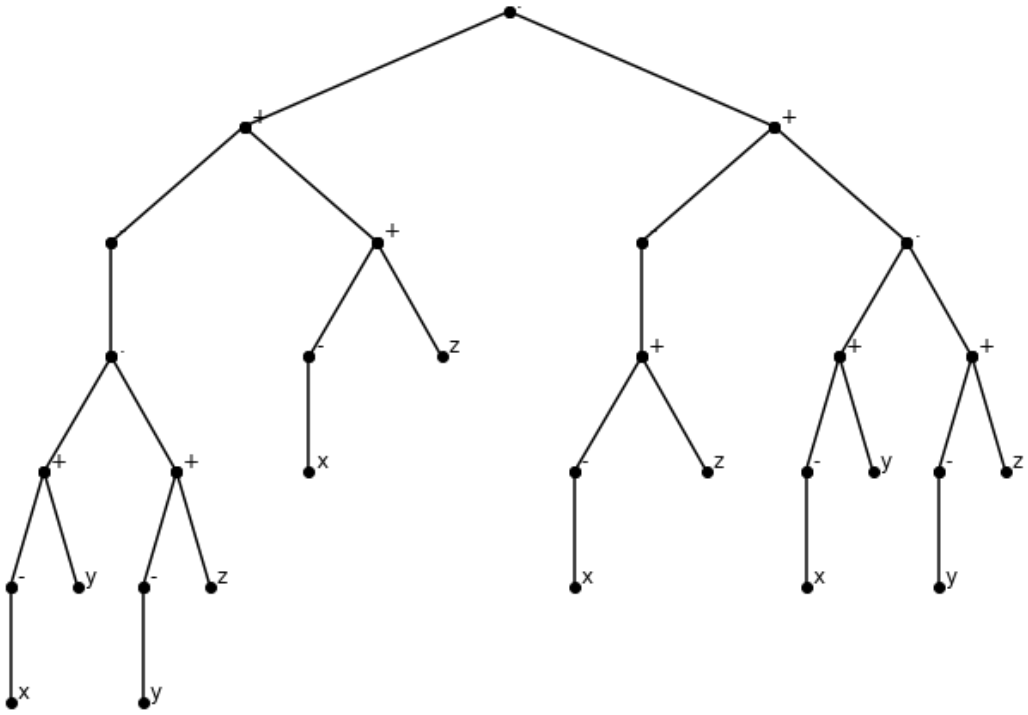
Voici une fonction qui prend en paramètre une formule f et renvoie une formule f' équivalente à f qui ne contient ni le connecteur $\rightarrow$ ni le connecteur $\leftrightarrow$. Comment faire cela ? Facile, en utilisant les équivalences classiques. On sait que $f_1 \rightarrow f_2 \equiv (-f_1 + f_2)$ et $f_1 \leftrightarrow f_2 \equiv (-f_1 + f_2).(-f_2 + f_1)$. Une fonction récursive s'impose.

In [46]:
```
def et_ou(f):
    if f[0] == 'var': return f
    elif f[0] == 'not': return ('not', et_ou(left(f)))
    else:
        f1 = et_ou(left(f))
        f2 = et_ou(right(f))
        if f[0] == 'and': return ('and', f1, f2)
        elif f[0] == 'or': return ('or', f1, f2)
        elif f[0] == 'imp': return ('or', ('not', f1), f2)
        elif f[0] == 'eqv': return ('and',('or', ('not', f1), f2),('or
```

In [47]:
```
to_string(et_ou(exemple))
```

Out[47]:
```
'(-((-x+y).(-y+z))+(-x+z)).(-(-x+z)+(-x+y).(-y+z))'
```

In [48]:
```
draw(et_ou(exemple))
```



3.2 La forme prénexe

Une formule est dite sous forme **prénexe** lorsque les seuls symboles de n'égation intervenant dans la formule le sont devant des variables. La fonction ci-dessous prend une formule f ne contenant ni "implique" ni "équivaut" et renvoie une formule équivalente à f sous forme prénexe. Comment faire ? Eh bien les lois de Morgan $-(f_1.f2) \equiv -f_1 + -f_2$ et $-(f_1 + f2) \equiv -f_1.-f_2$ ainsi que la loi $--f \equiv f$ sont nos amies ...

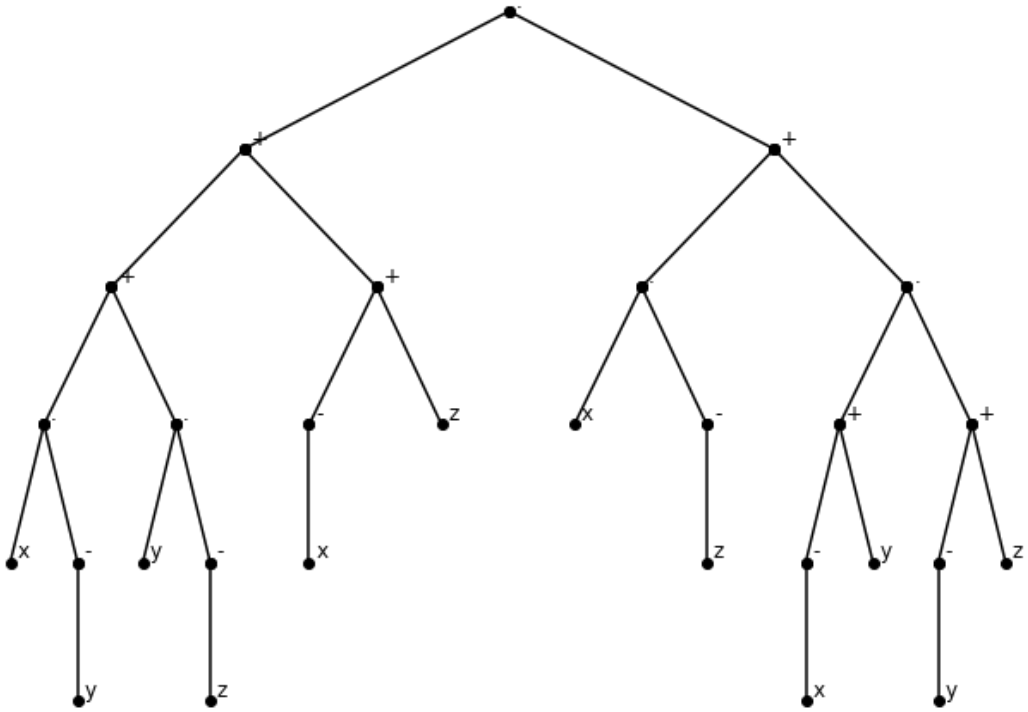
Remarque : à la fin de ce notebook nous serons en mesure de ***démontrer*** en Python que les lois de Morgan (et toutes les formules que nous voulons) sont des tautologies.

In [49]:
```
def prenexe(f):
    if f[0] == 'var': return f
    elif f[0] == 'not':
        f1 = left(f)
        if f1[0] == 'var': return ('not', f1)
        elif f1[0] == 'not': return prenexe(left(f1))
        else:
            g1 = prenexe(('not', left(f1)))
            g2 = prenexe(('not', right(f1)))
            if f1[0] == 'and': return ('or', g1, g2)
            else: return ('and', g1, g2)
    else:
        f1 = prenexe(left(f))
        f2 = prenexe(right(f))
        if f[0] == 'and': return ('and', f1, f2)
        else: return ('or', f1, f2)
```

In [50]:
```
to_string(prenexe(et_ou(exemple)))
```

Out[50]:
```
'((x.-y+y.-z)+(-x+z)).(x.-z+(-x+y).(-y+z))'
```

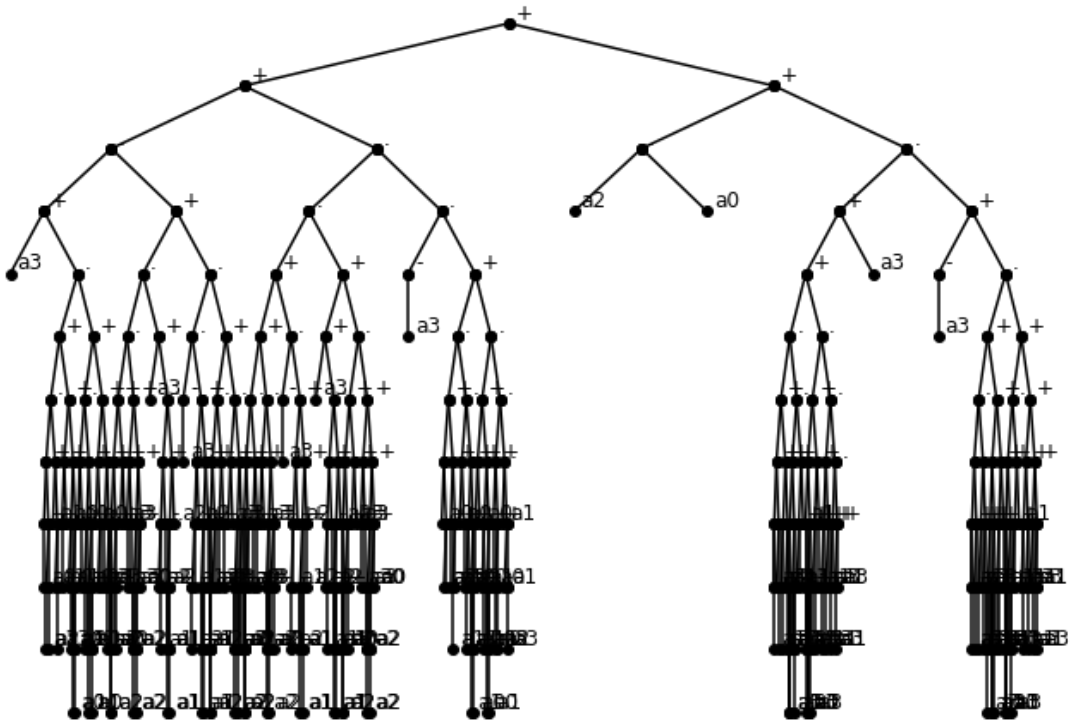
In [51]:
```
draw(prenexe(et_ou(exemple)))
```



In [52]:
```
f = prenexe(et_ou(formule_aleatoire(8, 4)))
to_string(f)
```

Out[52]:
```
'((a3+((-a0+(-a3+-a2)).(a1.(a1.-a3))+((-a0+a0)+(a1.-a0).(a1.-a0+a0.-
a1))).((a0.-a0).((-a1+a0)+(-a1+a0).(-a0+a1))+(a0.(a3.a2)+(-a1+(-a1+a
3))))).(((((-a2+a2).(-a2+a2)).(-a0+-a0)+(a3+a3)).(-a3.-a3+((a2.-a2+a
2.-a2)+a0.a0))).(a3+(a2.-a1+-a2).(((-a1+a1).(-a1+a1)).a2))+(-a3.((-a
2+a1).a2+((a1.-a1+a1.-a1)+-a2))).(((a2.-a2+a2.-a2)+a0.a0).(-a3.-a3)+
(a3+a3).(((-a2+a2).(-a2+a2)).(-a0+-a0))))+(((((a2.-a2+a2.-a2)+a0.a0)
.(-a3.-a3)+(a3+a3).(((-a2+a2).(-a2+a2)).(-a0+-a0)))+-a3.((-a2+a1).a2
+((a1.-a1+a1.-a1)+-a2))).((a3+(a2.-a1+-a2).(((-a1+a1).(-a1+a1)).a2))
+(((-a2+a2).(-a2+a2)).(-a0+-a0)+(a3+a3)).(-a3.-a3+((a2.-a2+a2.-a2)+a
0.a0)))).(-a3.((a0.(a3.a2)+(-a1+(-a1+a3))).((a0.-a0).((-a1+a0)+(-a1+
a0).(-a0+a1)))+((-a0+a0)+(a1.-a0).(a1.-a0+a0.-a1)).((-a0+(-a3+-a2)).
(a1.(a1.-a3))))))+(a2.a0+((((a3.-a1+a1.-a3)+(a3.-a1+a1.-a3)).(((-a3+
a3).(-a0+-a1)+(a0.a1).(a3.-a3)).((a3+a1).a1))+((a3.-a3+a0.a1).((-a0+
-a1)+(-a3+a3))+(-a3.-a1+-a1)).(((-a3+a1).(-a1+a3)).((-a3+a1).(-a1+a3
))))+a3).(-a3+(((-a3+a1).(-a1+a3)).((-a3+a1).(-a1+a3))+((a3.-a3+a0.a
1).((-a0+-a1)+(-a3+a3))+(-a3.-a1+-a1))).(((-a3+a3).(-a0+-a1)+(a0.a1)
.(a3.-a3)).((a3+a1).a1)+((a3.-a1+a1.-a3)+(a3.-a1+a1.-a3)))))'
```

In [53]:
```
draw(f)
```



3.3 Substitution

Soit f une formule. Soit x une variable. Soit g une autre formule. Comment substituer g à x dans la formule f ? Du code Python sera aussi clair qu'une longue explication.

In [54]:
```
def substituer(f, x, g):
    if f[0] == 'var':
        if f[1] == 'x': return g
        else: return f
    elif f[0] == 'not':
        return ('not', substituer(left(f), x, g))
    else:
        return (f[0], substituer(left(f), x, g), substituer(right(f), x
```

In [55]:
```
f = parse('x->(y->x)')
g = parse('a.b')
to_string(substituer(f, 'x', g))
```

Out[55]:
```
'a.b->(y->a.b)'
```

On pourrait continuer longtemps. Ce n'est évidemment pas la fin de l'histoire mais j'espère vous avoir montré que plus rien ne nous résiste concernant la **syntaxe** des formules. Nous allons passer maintenant à autre chose, la **sémantique**. Une formule est-elle vraie ou fausse ?

4. Sémantique: évaluation des formules

Disons-le tout de suite : une formule n'est NI vraie NI fausse. Faisons un parallèle avec les formules de l'algèbre. Considérons $A = x + y.z$, où $x, y, z \in \mathbb{R}$. Que vaut A ? C'est évidemment une question idiote. La réponse est : "ça dépend !". Oui, et de quoi ? Eh bien de x, y et z.

Maintenant si je vous dit que $x = 2$, $y = 3$ et $z = 4$, alors vous me direz que A vaut 15. Et vous aurez tort, ça vaut 14 :-). Bref, une formule ne vaut quelque chose que lorsque ses variables valent quelque chose. Pour évaluer une formule nous devons être dans un certain **environnement**. Revenons aux formules logiques et soyons précis :

Définition : un environnement est une fonction $\gamma : \mathcal{V} \to \{0, 1\}$.

Le choix de $\{0, 1\}$ est arbitraire. Moralement, 0 signifie "faux" et 1 signifie "vrai", mais toute autre **interprétation** est possible. Le mot est lâché. Sens ? Signification ? Interprétation ? Nous ne pouvons plus nous contenter d'écrire des formules, nous voulons en plus leur donner un sens. C'est le but de la **sémantique**.

Je n'irai pas plus avant dans la théorie. Pour faire court, on peut montrer que pour tout environnement γ, pour toute formule f, il existe une valeur dans $\{0, 1\}$ dépendant de f et γ, pour laquelle les connecteurs apparaissant dans f ont le comportement que l'on souhaite. On note cette valeur $eval(f, \gamma)$. Par exemple, il est souhaitable que $eval(f_1.f_2, \gamma)$ soit égal à $eval(f_1, \gamma) \times eval(f_2, \gamma)$ pour repecter ce que l'on pense du "et" (faux et faux = faux, faux et vrai = faux, etc.).

On montre également que cette valeur ne dépend que de la valeur de γ en les variables qui apparaissent dans f.

4.1 Codes de longueur n

Étape 1 : fabriquer tous les n-uplets de 0 et de 1, pour un n donné.

In [56]:
```
def codes(n):
    if n == 0: return [[]]
    else:
        cs = codes(n - 1)
        return [[0] + c for c in cs] + [[1] + c for c in cs]
```

In [57]:
```
codes(4)
```

Out[57]:
```
[[0, 0, 0, 0],
 [0, 0, 0, 1],
 [0, 0, 1, 0],
 [0, 0, 1, 1],
 [0, 1, 0, 0],
 [0, 1, 0, 1],
 [0, 1, 1, 0],
 [0, 1, 1, 1],
 [1, 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]]
```

4.2 Environnements

Étape 2 : fabriquer tous les environnements possibles pour une liste donnée de variables.

In [58]:
```
def environnement(vs, c):
    gamma = {}
    for i in range(len(vs)):
        gamma[vs[i]] = c[i]
    return gamma
```

In [59]:
```
environnement(['x', 'y', 'z'], [0, 1, 0])
```

Out[59]:
```
{'x': 0, 'y': 1, 'z': 0}
```

In [60]:
```
def environnements(vs):
    n = len(vs)
    cs = codes(n)
    return [environnement(vs, c) for c in cs]
```

In [61]:
```
environnements(list(variables(exemple)))
```

Out[61]:
```
[{'x': 0, 'y': 0, 'z': 0},
 {'x': 0, 'y': 1, 'z': 0},
 {'x': 1, 'y': 0, 'z': 0},
 {'x': 1, 'y': 1, 'z': 0},
 {'x': 0, 'y': 0, 'z': 1},
 {'x': 0, 'y': 1, 'z': 1},
 {'x': 1, 'y': 0, 'z': 1},
 {'x': 1, 'y': 1, 'z': 1}]
```

3.3 Évaluer une formule dans un environnement

Nous y voilà. Voici la fonction d'évaluation. La comprendre, c'est la lire. Les formules pour l'implication et l'équivalence peuvent paraître absconses, vérifiez-les.

In [62]:
```
def eval(f, gamma):
    if f[0] == 'var': return gamma[f[1]]
    elif f[0] == 'not':
        b = eval(f[1], gamma)
        return 1 - b
    else:
        b1 = eval(f[1], gamma)
        b2 = eval(f[2], gamma)
        if f[0] == 'and': return b1 * b2
        elif f[0] == 'or': return b1 + b2 - b1 * b2
        elif f[0] == 'imp': return 1 - b1 + b1 * b2
        elif f[0] == 'eqv': return 1 - b1 - b2 + 2 * b1 * b2
```

Voici l'évaluation de notre exemple fétiche sur tous les environnements possibles.

In [63]:
```
env = environnements(list(variables(exemple)))
[(gamma, eval(exemple, gamma)) for gamma in env]
```

Out[63]:
```
[({'x': 0, 'y': 0, 'z': 0}, 1),
 ({'x': 0, 'y': 1, 'z': 0}, 0),
 ({'x': 1, 'y': 0, 'z': 0}, 1),
 ({'x': 1, 'y': 1, 'z': 0}, 1),
 ({'x': 0, 'y': 0, 'z': 1}, 1),
 ({'x': 0, 'y': 1, 'z': 1}, 1),
 ({'x': 1, 'y': 0, 'z': 1}, 0),
 ({'x': 1, 'y': 1, 'z': 1}, 1)]
```

3.4 Table de vérité

La fonction ci-dessous ne fait qu'afficher sous forme plus "lisible" le résultat que nous avons obtenu ci-dessus. On range tout cela dans ce que l'on appelle "la" table de vérité de la formule.

In [64]:
```
def afficher_table(f):
    vars = list(variables(f))
    vars.sort()
    env = environnements(vars)
    for x in env[0]: sys.stdout.write('%3s' % x) # étiquettes de la ta
    sys.stdout.write('%3s\n'% 'f') # la dernière étiquette
    for gamma in env:
        b = eval(f, gamma)
        for x in gamma: sys.stdout.write('%3d' % gamma[x])
        sys.stdout.write('%3d\n' % b)
```

In [65]:
```
afficher_table(exemple)
```

```
  x  y  z  f
  0  0  0  1
  0  0  1  1
  0  1  0  0
  0  1  1  1
  1  0  0  1
  1  0  1  0
  1  1  0  1
  1  1  1  1
```

In [66]:
```
afficher_table(formule_aleatoire(10, 4))
```

```
 a0 a1 a2 a3  f
  0  0  0  0  1
  0  0  0  1  1
  0  0  1  0  1
  0  0  1  1  1
  0  1  0  0  0
  0  1  0  1  0
  0  1  1  0  0
  0  1  1  1  0
  1  0  0  0  0
  1  0  0  1  0
  1  0  1  0  1
  1  0  1  1  1
  1  1  0  0  0
  1  1  0  1  0
  1  1  1  0  0
  1  1  1  1  0
```

3.5 Satisfiabilité, tautologies, contradictions

Ce notebook est déjà bien long. Je me contenterai de finir en disant deux mots sur la satisfiabilité.

Définition : soient f une formule et γ un environnement. On dit que γ satisfait f (ou que f est satisfaite par γ) lorsque $eval(f, \gamma) = 1$.

Définition : Une formule est satisfiable lorsqu'il existe un environnement qui la satisfait.

Définition : Une formule est une contradiction lorsqu'aucun environnement ne la satisfait.

Définition : Une formule est une tautologie lorsque tous les environnements la satisfont.

Une tautologie est donc une formule très satisfaite :-).

In [67]:
```
def satisfiable(f):
    vars = list(variables(f))
    env = environnements(vars)
    for gamma in env:
        if eval(f, gamma) == 1: return (True, gamma)
    return (False, None)
```

In [68]:
```
f = parse(('x->(y->x)'))
satisfiable(f)
```

Out[68]:
```
(True, {'x': 0, 'y': 0})
```

In [69]:
```
def contradiction(f):
    return not satisfiable(f)
```

In [70]:
```
def tautologie(f):
    vars = list(variables(f))
    env = environnements(vars)
    for gamma in env:
        if eval(f, gamma) == 0: return False
    return True
```

In [71]:
```
tautologie(f)
```

Out[71]:
```
True
```

In [72]:
```
f = formule_aleatoire(10, 5)
tautologie(f)
```

Out[72]:
```
False
```

Et notre exemple adoré ? Est-ce une tautologie ? Est-il au moins satisfiable ?

In [73]:
```
satisfiable(exemple)
```

Out[73]:
```
(True, {'x': 0, 'y': 0, 'z': 0})
```

In [74]:
```
tautologie(exemple)
```

Out[74]:
```
False
```

3.5 Tautologies incontournables

Une des tâches première du logicien en particulier et du mathématicien en général est d'écrire des tautologies. Parmi les tautologies fondamentales de la logique nous trouvons :

3.5.1 La double négation

In [75]:
```
double_negation = parse('--x<->x')
tautologie(double_negation)
```

Out[75]:
```
True
```

3.5.2 La non-contradiction

In [76]:
```
non_contradiction = parse('-(x.-x)')
tautologie(non_contradiction)
```

Out[76]:
```
True
```

3.5.3 Le tiers exclu

In [77]:
```
tiers_exclu = parse('x+-x')
tautologie(tiers_exclu)
```

Out[77]:
```
True
```

3.5.4 Les lois de Morgan

In [78]:
```
morgan1 = parse('-(a.b)<->-a+-b')
tautologie(morgan1)
```

Out[78]:
```
True
```

In [79]:
```
morgan2 = parse('-(a+b)<->-a.-b')
tautologie(morgan2)
```

Out[79]: True

3.5.5 Le modus Ponens

In [80]:
```
modus_ponens = parse('x.(x->y)->y')
tautologie(modus_ponens)
```

Out[80]: True

3.5.6 La transitivité de l'implication

In [81]:
```
trans_impl = parse('(x->y).(y->z)->(x->z)')
tautologie(trans_impl)
```

Out[81]: True

3.5.7 Le principe de contraposition

In [82]:
```
contra = parse('(a->b)<->(-b->-a)')
tautologie(contra)
```

Out[82]: True

Et bien d'autres ... démontrez autant que vous voudrez. Quoique "démontrer" veuille maintenant dire "appuyer sur la touche Entrée" :-)

In [ ]:
```

```