

Informatique et Programmation

Cours 12

Jean-Jacques Lévy

`jean-jacques.levy@inria.fr`

`http://jeanjacqueslevy.net/prog-py`

Plan

- classes et objets (rappel)
- représentation des arbres
- ordres de parcours d'arbre
- belle impression
- arbres de syntaxe abstraite

dès maintenant: **télécharger Python 3 en** `http://www.python.org`

Arbres (représentation 1)

- on définit une classe avec des champs et des méthodes

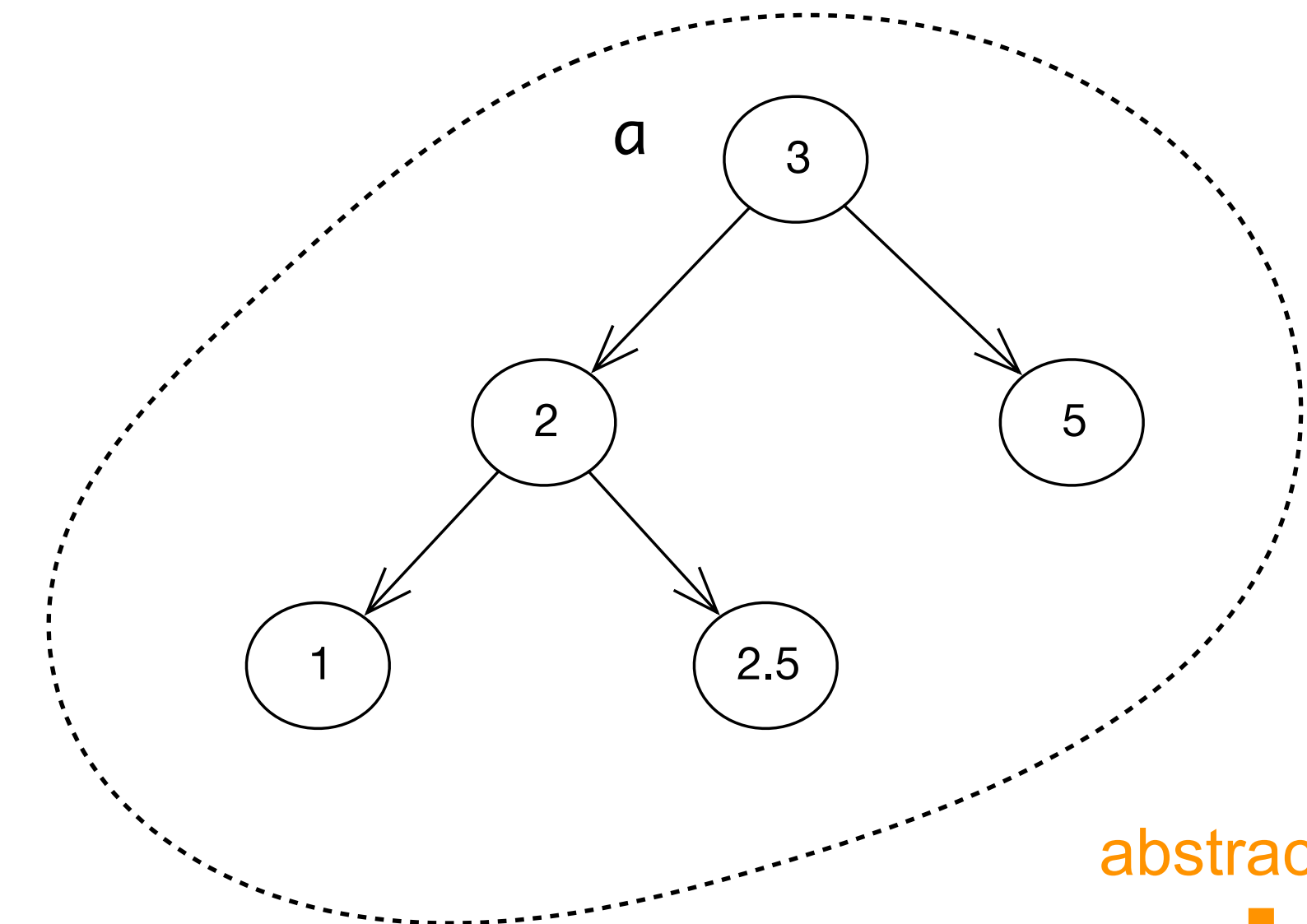
```
class Noeud:  
    def __init__(self, x, g, d) :  
        self.val = x  
        self.gauche = g  
        self.droit = d
```

- on définit une classe pour les feuilles

```
class Feuille:  
    def __init__(self, x) :  
        self.val = x
```

- et on construit des arbres

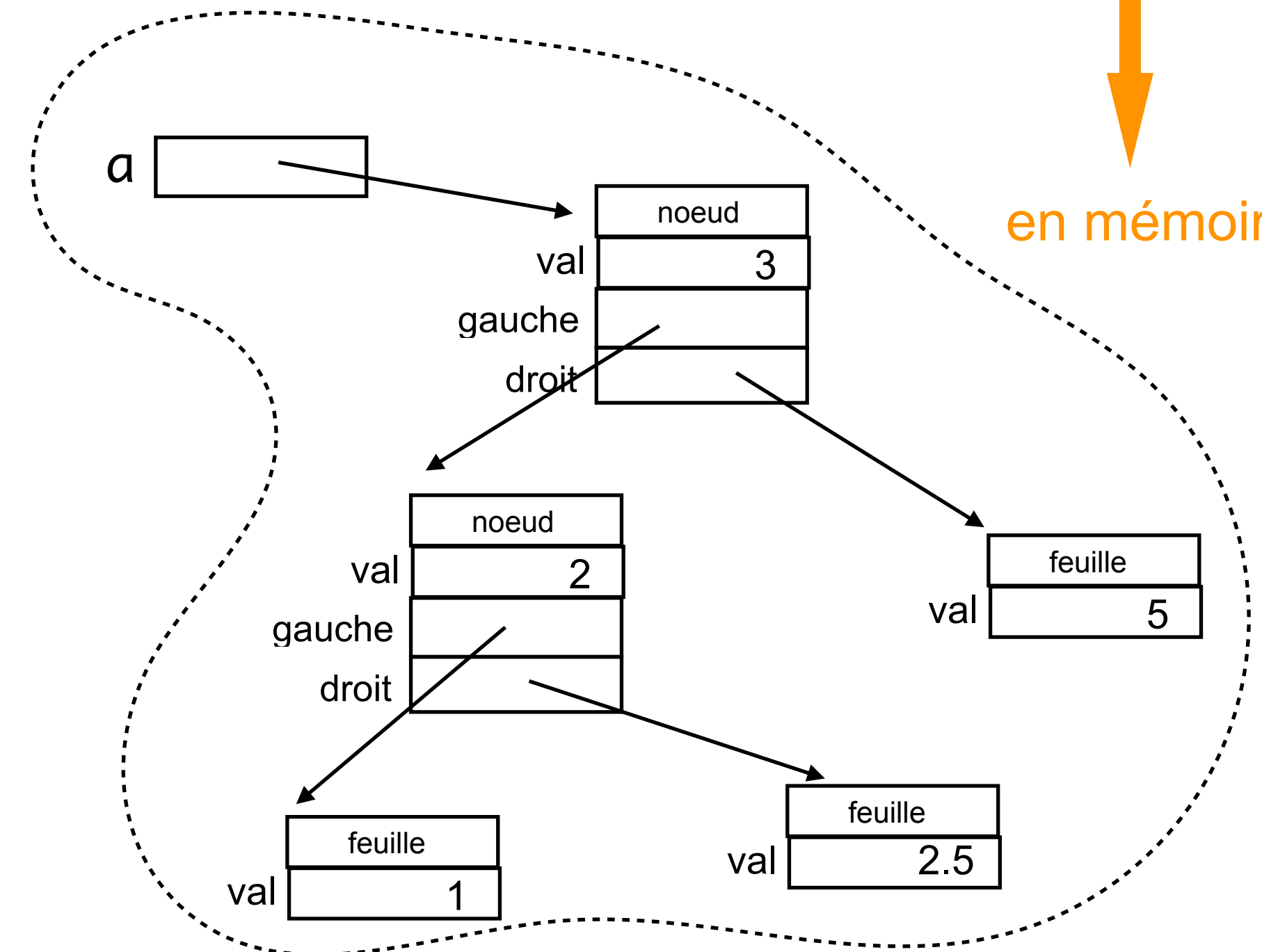
```
a = Noeud (3, Noeud (2, Feuille (1), Feuille (2.5)), Feuille (5))
```



abstraction



en mémoire



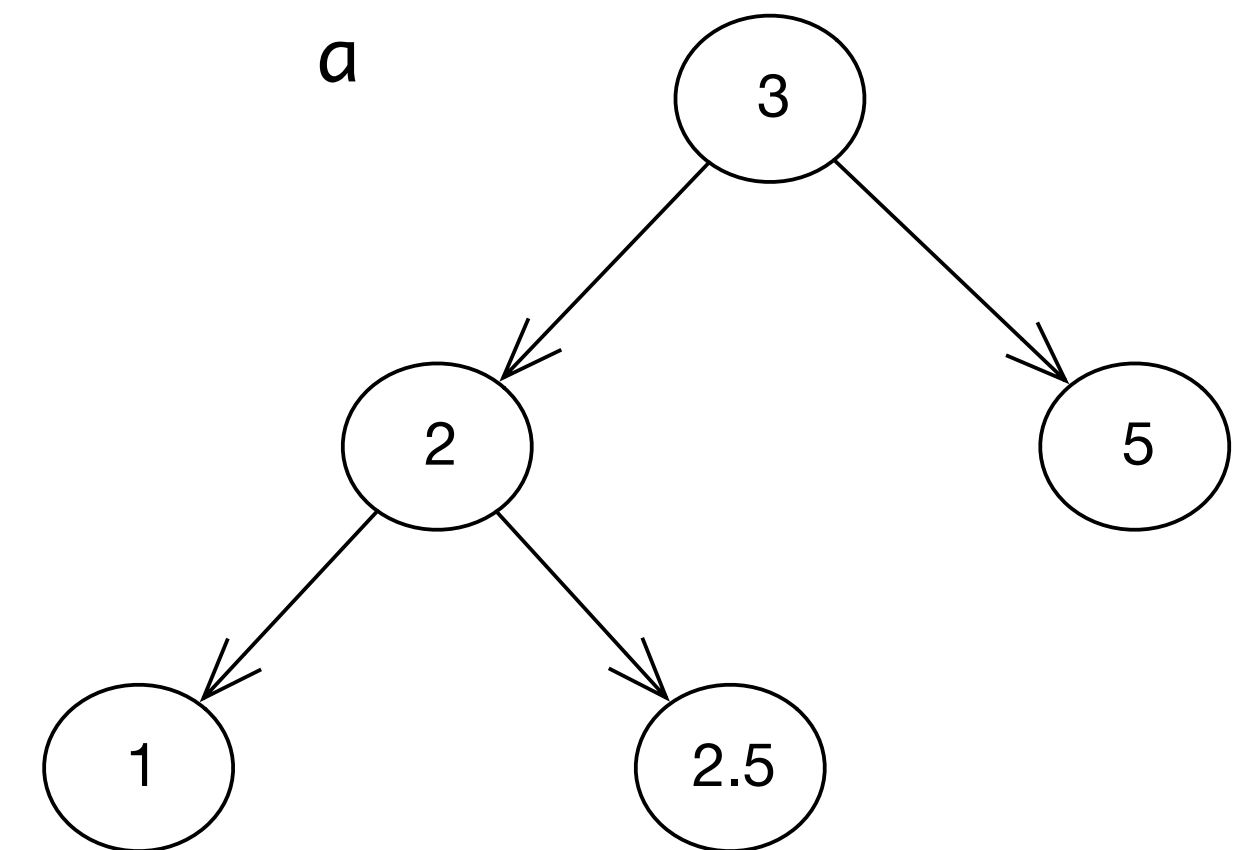
Arbres

- on définit une méthode pour l'impression des noeuds et des feuilles

```
class Noeud:  
    # comme avant  
  
    def __str__ (self) :  
        return "Noeud ({{}, {{}, {{})".format (self.val, self.gauche, self.droit)  
  
class Feuille:  
    # comme avant  
  
    def __str__ (self) :  
        return "Feuille ({{})".format (self.val)
```

- on construit et imprime des arbres

```
a = Noeud (3, Noeud (2, Feuille (1), Feuille (2.5)), Feuille (5))  
  
print (a)  
➔ Noeud (3, Noeud (2, Feuille (1), Feuille (2.5)), Feuille (5))  
  
print (a.droit)  
➔ Feuille (5)  
  
print (a.gauche)  
➔ Noeud (2, Feuille (1), Feuille (2.5))
```



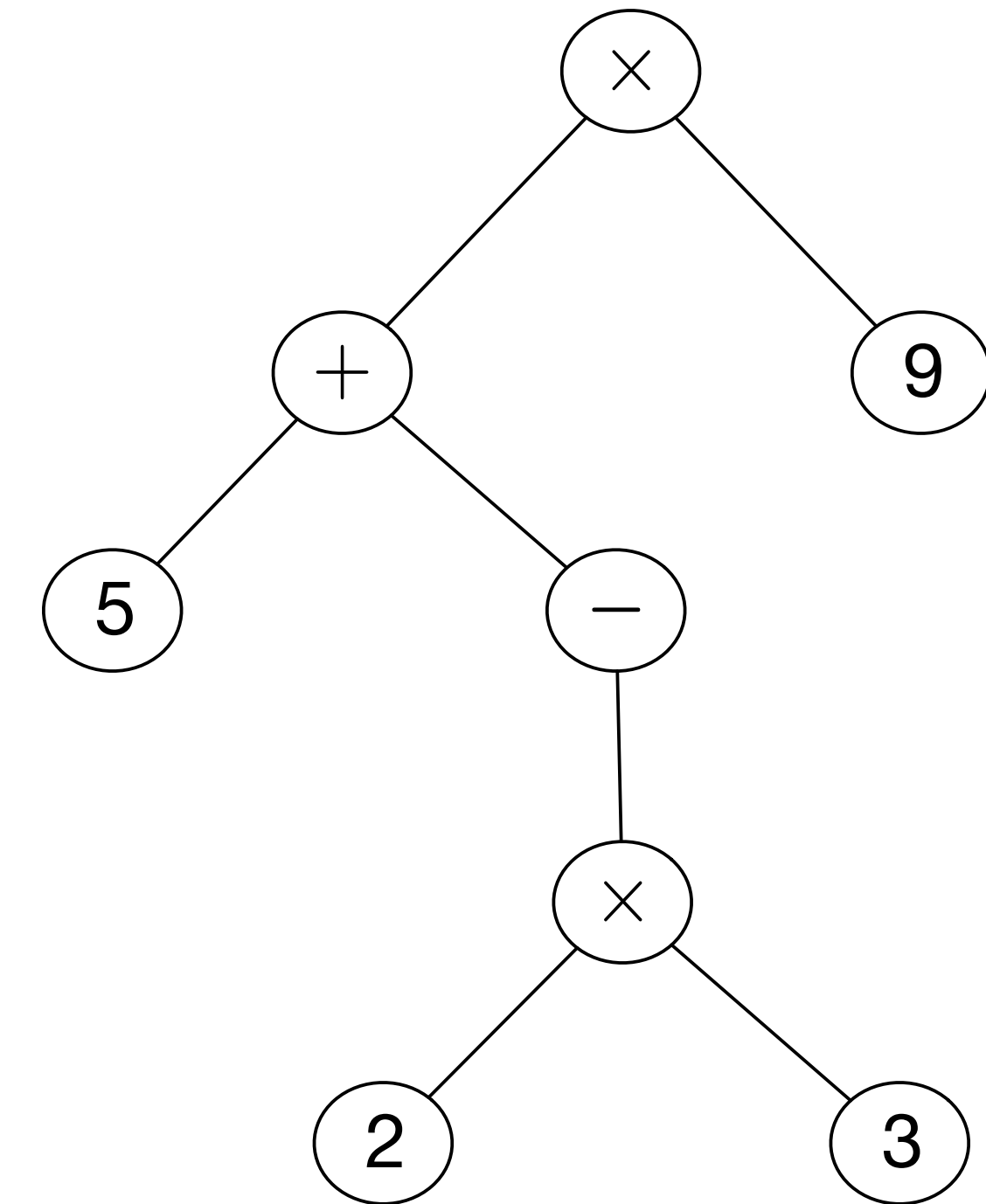
Arbres (représentation 2)

- On peut distinguer les noeuds binaires et les noeuds unaires

```
class Noeud_Bi:  
    def __init__(self, x, g, d) :  
        self.val = x  
        self.gauche = g  
        self.droit = d
```

```
class Noeud_Un:  
    def __init__(self, x, a) :  
        self.val = x  
        self.fils = a
```

```
class Feuille:  
    def __init__(self, x) :  
        self.val = x
```



- et on construit l'arbre par:

```
d = Noeud_Bi ('*',  
            Noeud_Bi ('+', Feuille (5),  
                    Noeud_Un ('-',  
                              Noeud_Bi ('*', Feuille (2), Feuille (3)))),  
            Feuille (9))
```

Arbres (représentation 3)

- None représente l'arbre vide (0 neuds, 0 feuilles)

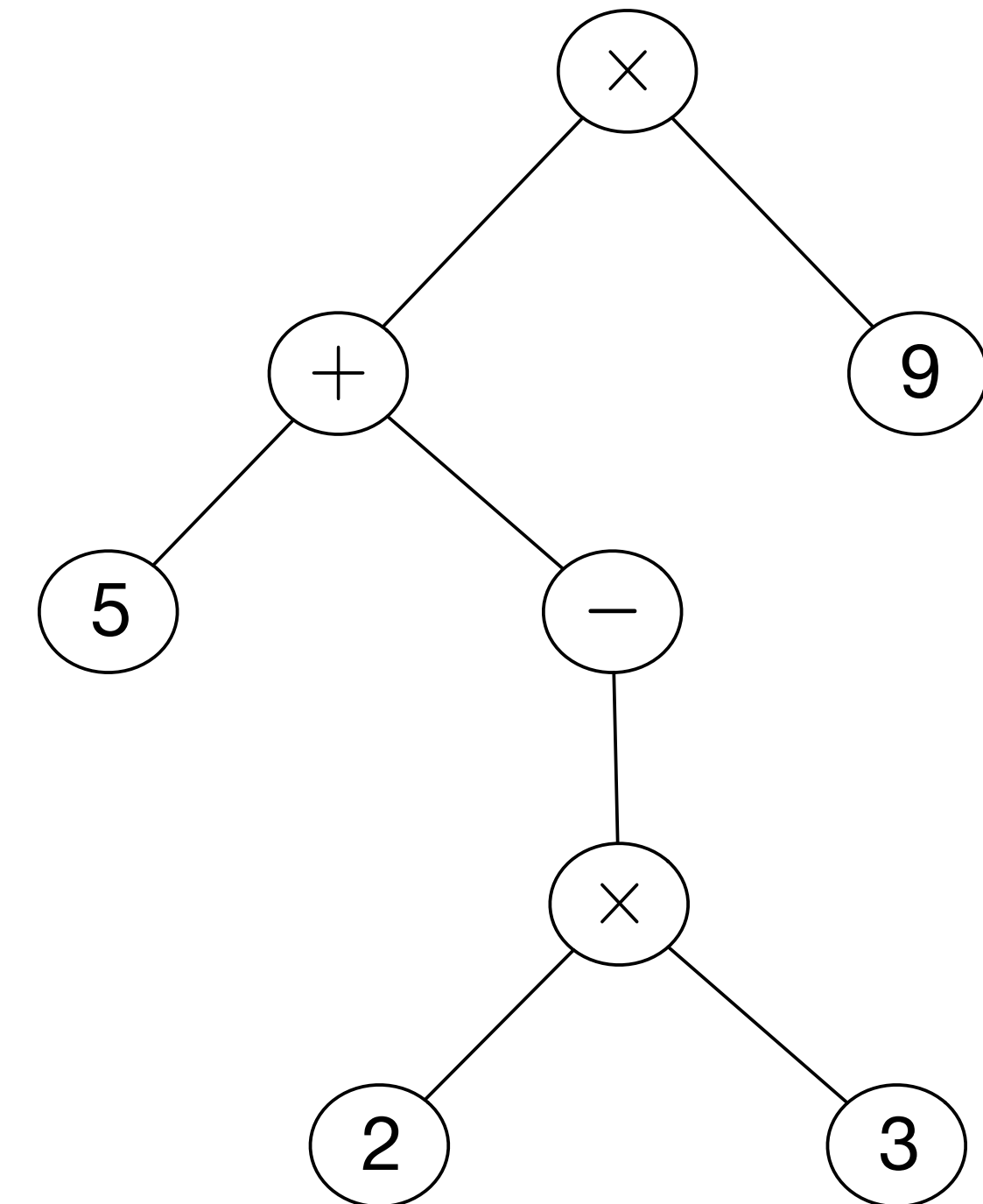
```
c = Noeud ('*', Noeud ('+', Feuille (5),  
                    Noeud ('-', Noeud ('*', Feuille (2), Feuille (3)),  
                    None)),  
        Feuille (9))
```

- ou encore ici :

```
c = Noeud ('*', Noeud ('+', Feuille (5),  
                    Noeud ('-', None,  
                    Noeud ('*', Feuille (2), Feuille (3)))),  
        Feuille (9))
```

- ou encore en identifiant feuilles et noeuds sans fils

```
c = Noeud ('*', Noeud ('+', Noeud (5, None, None),  
                    Noeud ('-', None,  
                    Noeud ('*', Noeud (2, None, None), Noeud (3, None, None)))),  
        Noeud (9, None, None))
```



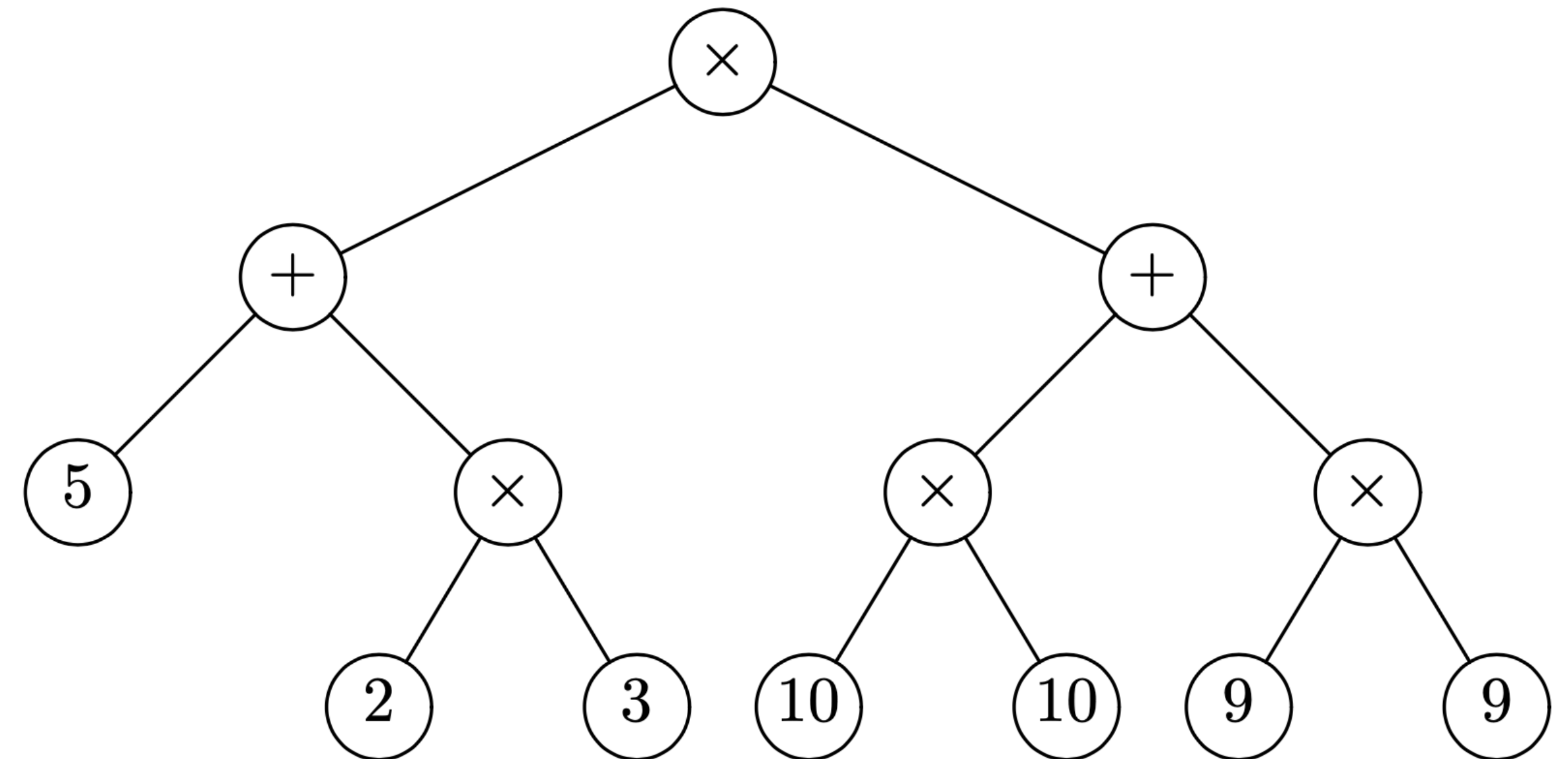
Arbres (et fonctions récursives)

← induction structurelle

- On parcourt ou calcule sur les arbres avec des fonctions récursives

```
def hauteur (a) :  
    if isinstance (a, Feuille) :  
        return 0  
    else :  
        return 1 + max (hauteur (a.gauche), hauteur (a.droit))
```

```
def taille (a) :  
    if isinstance (a, Feuille) :  
        return 1  
    else :  
        return 1 + taille (a.gauche) + taille (a.droit)
```



- et on calcule les hauteur et taille

```
print (b)  
Noeud (*, Noeud (+, Feuille (5), Noeud (*, Feuille (2), Feuille (3))), Noeud (+, Noeud (*, Feuille (10),  
Feuille (10)), Noeud (*, Feuille (9), Feuille (9))))
```

```
print (hauteur (b))  
3
```

```
print (taille (b))  
13
```

Arbres (et méthodes)

- On parcourt ou calcule sur les arbres avec des méthodes

```
class Noeud:  
    # comme avant et en plus :  
    def hauteur (self) :  
        return 1 + max (self.gauche.hauteur(), self.droit.hauteur())  
  
    def taille (self) :  
        return 1 + a.gauche.taille() + a.droit.taille()
```

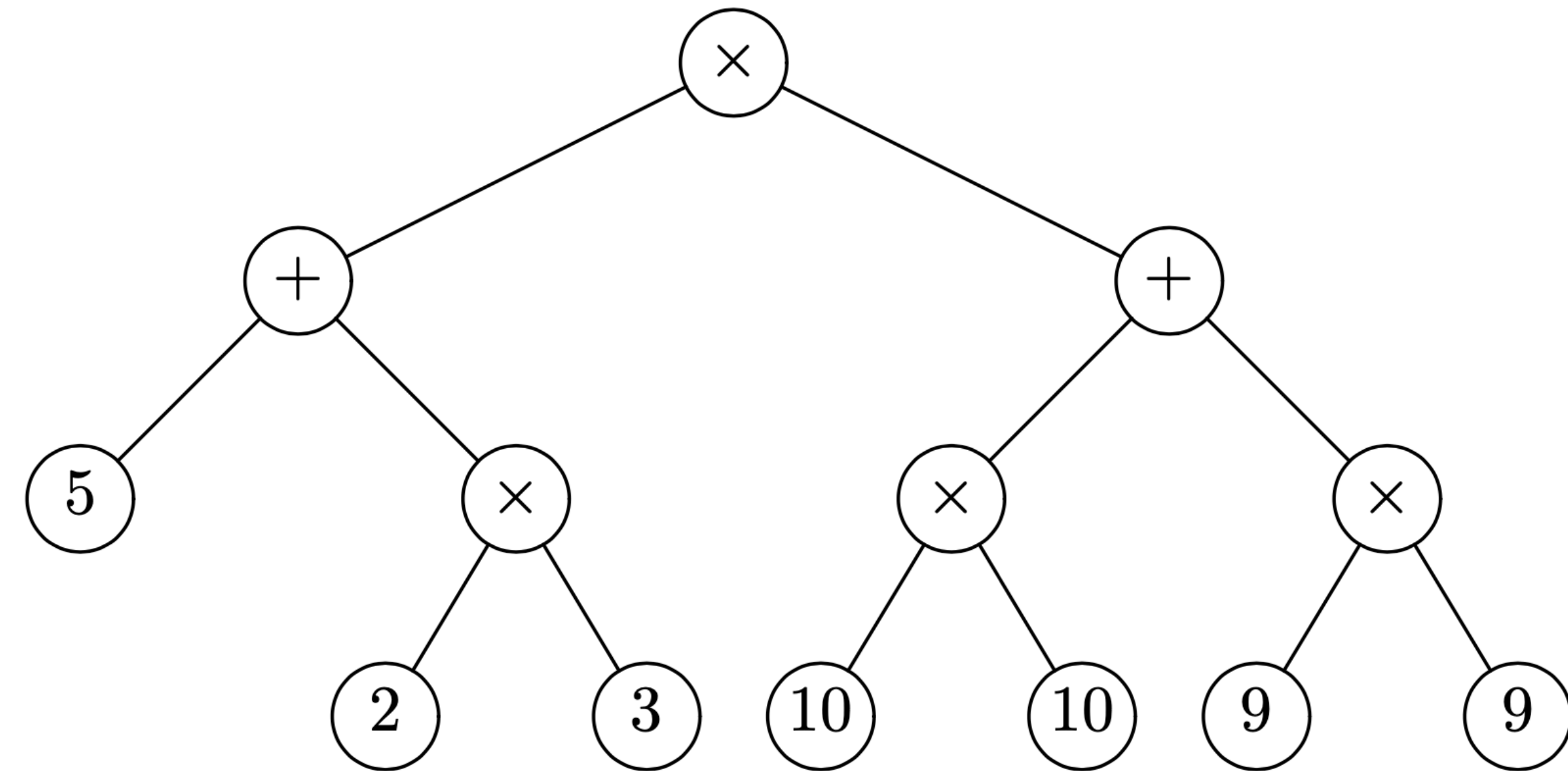
```
class Feuille:  
    # comme avant et en plus :  
    def hauteur (self) :  
        return 0  
  
    def taille (self) :  
        return 1
```

- et on calcule les hauteur et taille

```
print (b.hauteur())  
3
```

```
print (b.taille())  
13
```

← par cas sur sous-classes



Arbres

- choisir entre fonctions récursives et méthodes est affaire de goût
- les fonctions récursives favorisent la programmation procédurale
[**tout est fonction (procédure)**]
- les méthodes privilégient la programmation par objets
[**tout est piloté par les données**]

Arbres (représentation 4)

- une arborescence est une représentation par lien arrière

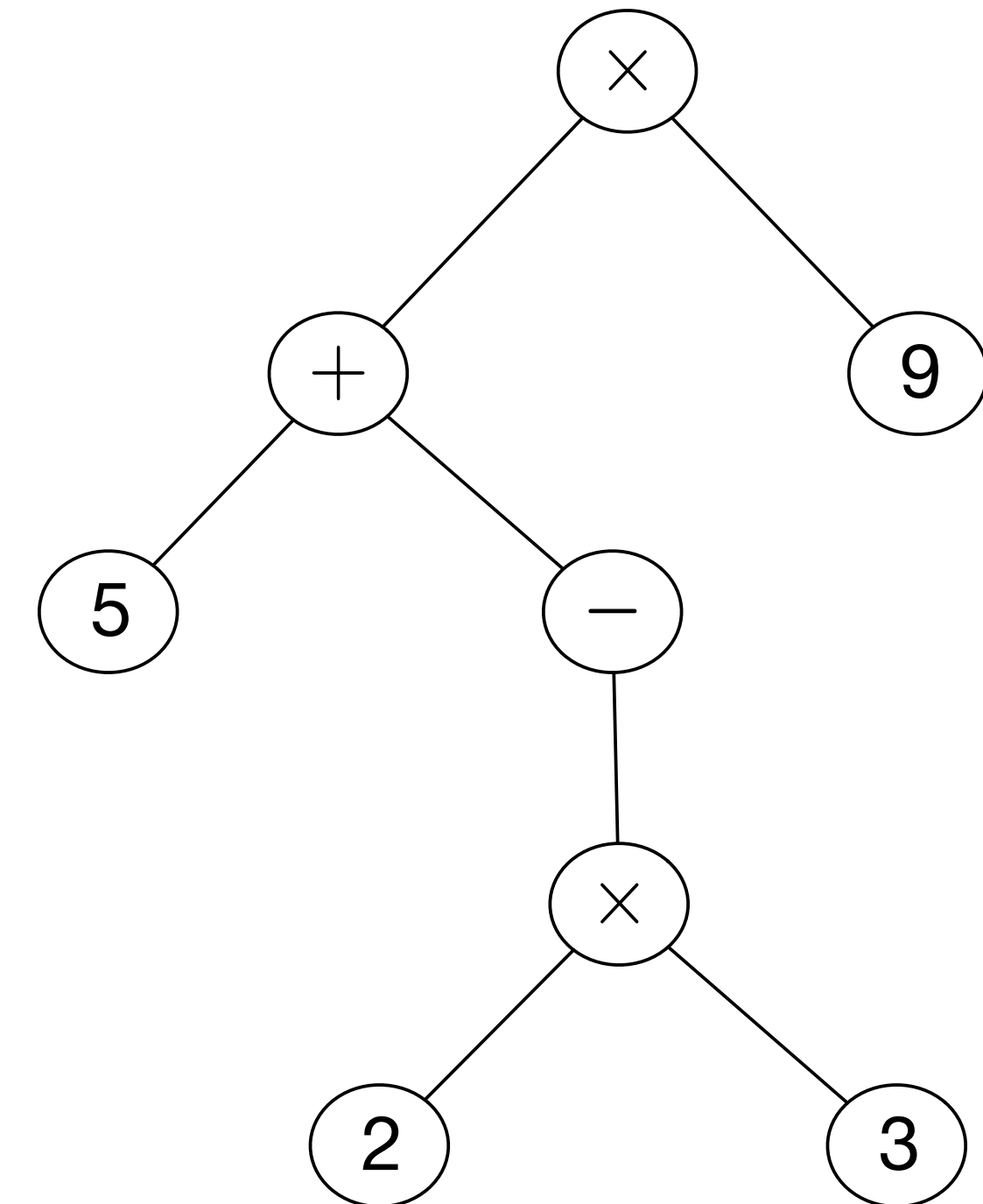
```
arbre = [ ('x', None),  
          ('+', 0),  
          ('5', 1),  
          ('-', 1),  
          ('x', 3),  
          ('2', 4),  
          ('3', 4),  
          ('9', 4)]
```

- ou encore

```
val = ['x', '+', '5', '-', 'x', '2', '3', '9']  
pere = [None, 0, 1, 1, 3, 4, 4, 0]
```

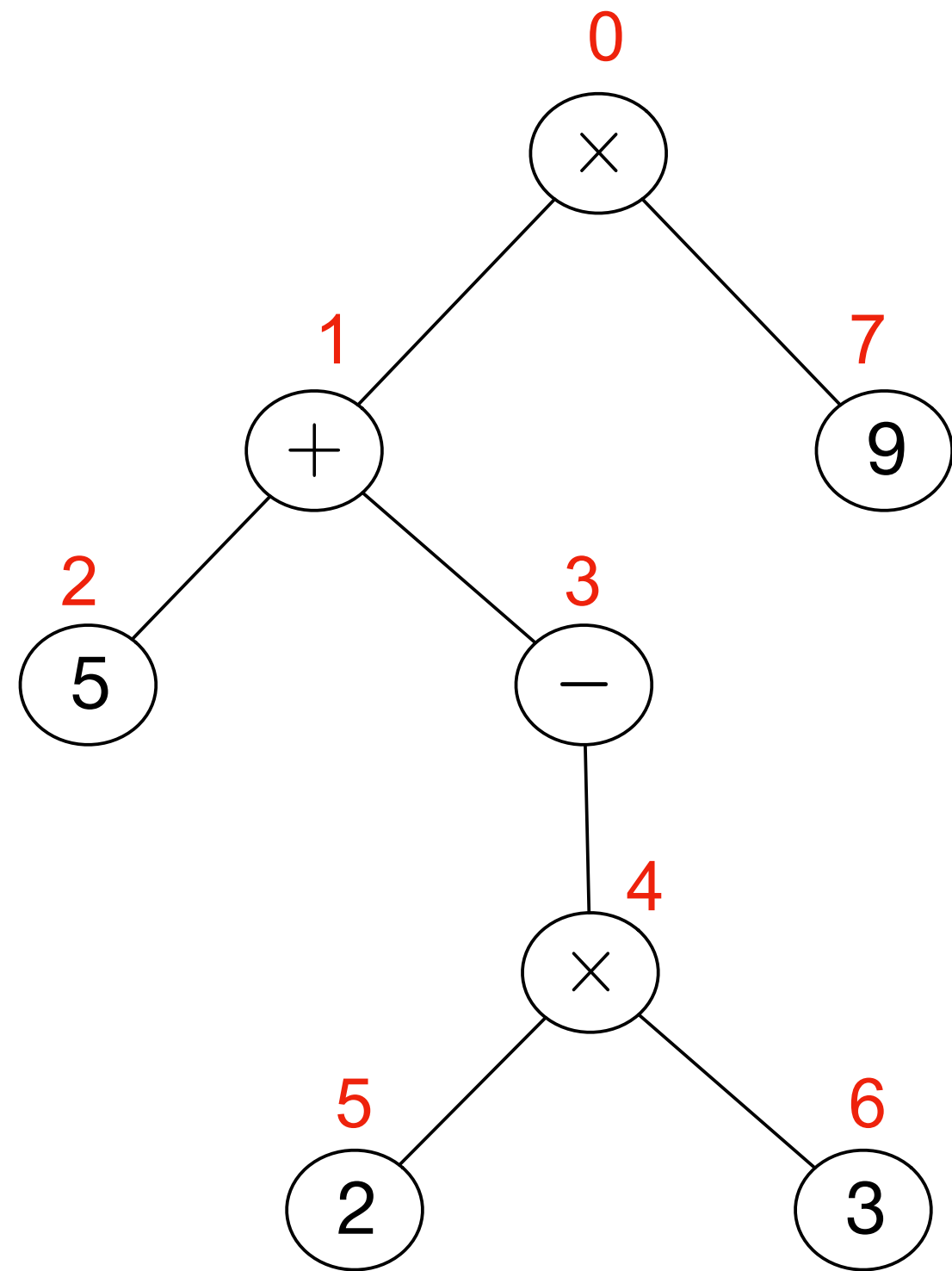
- la représentation plus souple car ne distinguant pas l'arité des noeuds
- mais elle ne permet pas un simple parcours d'arbre

Exercice Passer de la représentation 1 à la représentation 4 et réciproquement.



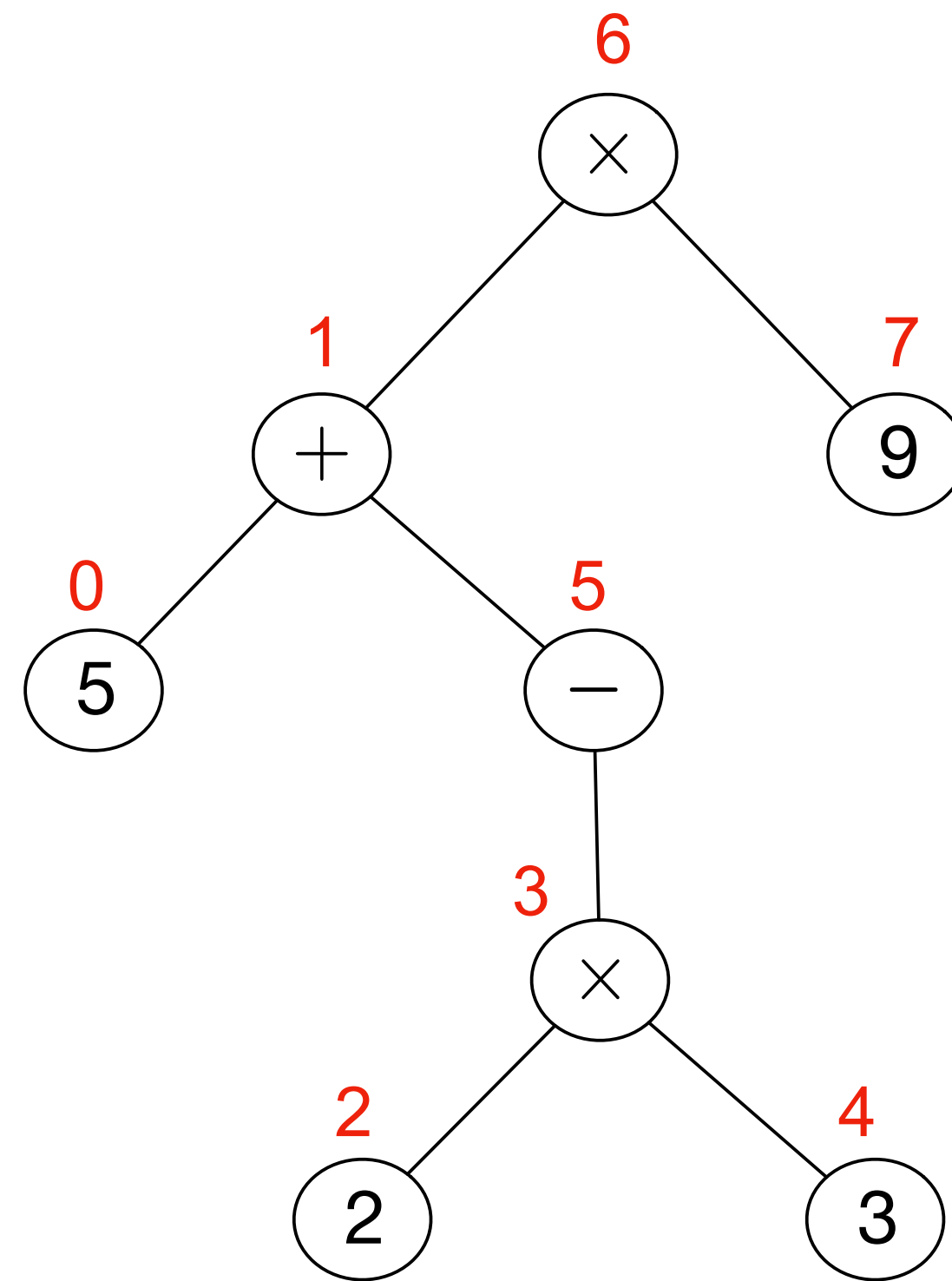
Parcours d'arbre

- 3 parcours d'arbre (préfixe, infixe, postfixe)



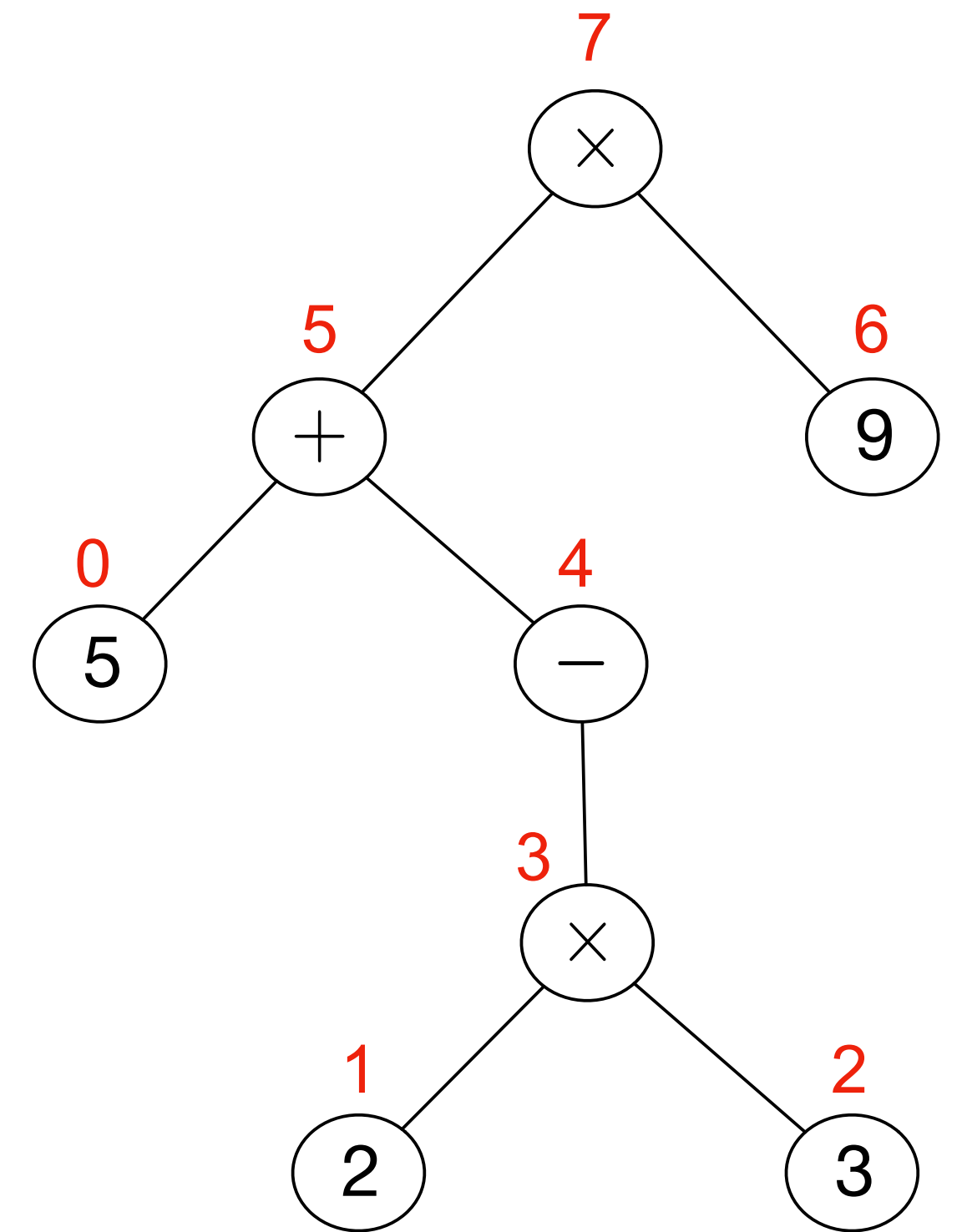
préfixe

- notation polonaise préfixe
 $x + 5 - x 2 3 9$



infixe

- notation arithmétique
 $(5 + - (2 \times 3)) \times 9$



postfixe

- notation polonaise postfixe
 $5 2 3 x - + 9 x$

Parcours d'arbre

- générer les notations préfixe, postfixe et infixe

```
def polprefix (a) :  
  if isinstance (a, Feuille) :  
    return a.val  
  elif isinstance (a, Noeud_Un) :  
    return a.val + ' ' + polprefix (a.fils)  
  else :  
    return a.val \  
      + ' ' + polprefix (a.gauche) \  
      + ' ' + polprefix (a.droit)
```

```
def polpostfix (a) :  
  if isinstance (a, Feuille) :  
    return a.val  
  elif isinstance (a, Noeud_Un) :  
    return polpostfix (a.fils) + ' ' + a.val  
  else :  
    return polpostfix (a.gauche) \  
      + ' ' + polpostfix (a.droit) \  
      + ' ' + a.val
```

```
def notinfixe (a) :  
  if isinstance (a, Feuille) :  
    return a.val  
  elif isinstance (a, Noeud_Un) :  
    return '(' + a.val + ' ' + notinfixe (a.fils) + ')'  
  else :  
    return '(' + notinfixe (a.gauche) \  
      + ' ' + a.val \  
      + ' ' + notinfixe (a.droit) + ')'
```

← trop de parenthèses
[on verra plus tard pour les enlever]

- notation polonaise préfixe

x + 5 - x 2 3 9

- notation infixe

(5 + - (2 x 3)) x 9

- notation polonaise postfixe

5 2 3 x - + 9 x

Arbres (représentation 5)

- Arbres n-aires avec nombre arbitraire de fils

```
class Noeud:
    def __init__(self, x, l) :
        self.val = x
        self.fils = l
    #
    def __str__(self) :
        r = ''
        for a in self.fils :
            r = r + ', ' + str(a)
        return "Noeud ({}).format (r[2:])"
```



```
def __str__(self) :
    r = ', '.join(map(str, self.fils))
    return "Noeud ({}).format (r)"
```

```
class Feuille:
    def __init__(self, x) :
        self.val = x
    #
    def __str__(self) :
        return "Feuille ({}).format (self.val)"
```

```
a = Noeud (10,
           [Noeud (12, [Feuille (3)]),
            Feuille (4), Feuille (5)])
print (a)
```

Arbres - Belle impression

- on peut réduire le nombre de parenthèses si on connaît la précedence des opérateurs

- en mathématiques, ‘*’ a une plus forte précedence que ‘+’

$$3 + 4 \times 5 \quad \equiv \quad 3 + (4 \times 5)$$

$$(11 + 3) * 4 \quad \not\equiv \quad 11 + 3 * 4$$

- on peut donc faire le dictionnaire suivant de précedences: $\text{preds} = \{ '+' : 0, '*' : 2, '-' : 3 \}$

- la fonction d'impression met des parenthèses si la précedence est inférieure à la précedence du contenant

Arbres - Belle impression

- on peut réduire le nombre de parenthèses si on connaît la précedence des opérateurs

```
preds = {'+': 0, '*': 2, '-': 3}
```

```
def notifix (a, p) :  
    if isinstance (a, Feuille) :  
        return a.val  
    elif isinstance (a, Noeud_Un) :  
        q = preds[a.val]  
        if p > q :  
            return '(' + ' ' + a.val \  
            + ' ' + notifix (a.fils, q) + ')'  
        else :  
            return a.val + ' ' + notifix (a.fils, q)  
    else :  
        q = preds[a.val]  
        if p > q :  
            return '(' + notifix (a.gauche, q) \  
            + ' ' + a.val \  
            + ' ' + notifix (a.droit, q) + ')'  
        else :  
            return notifix (a.gauche, q) \  
            + ' ' + a.val \  
            + ' ' + notifix (a.droit, q)
```

Arbres de syntaxe abstraite

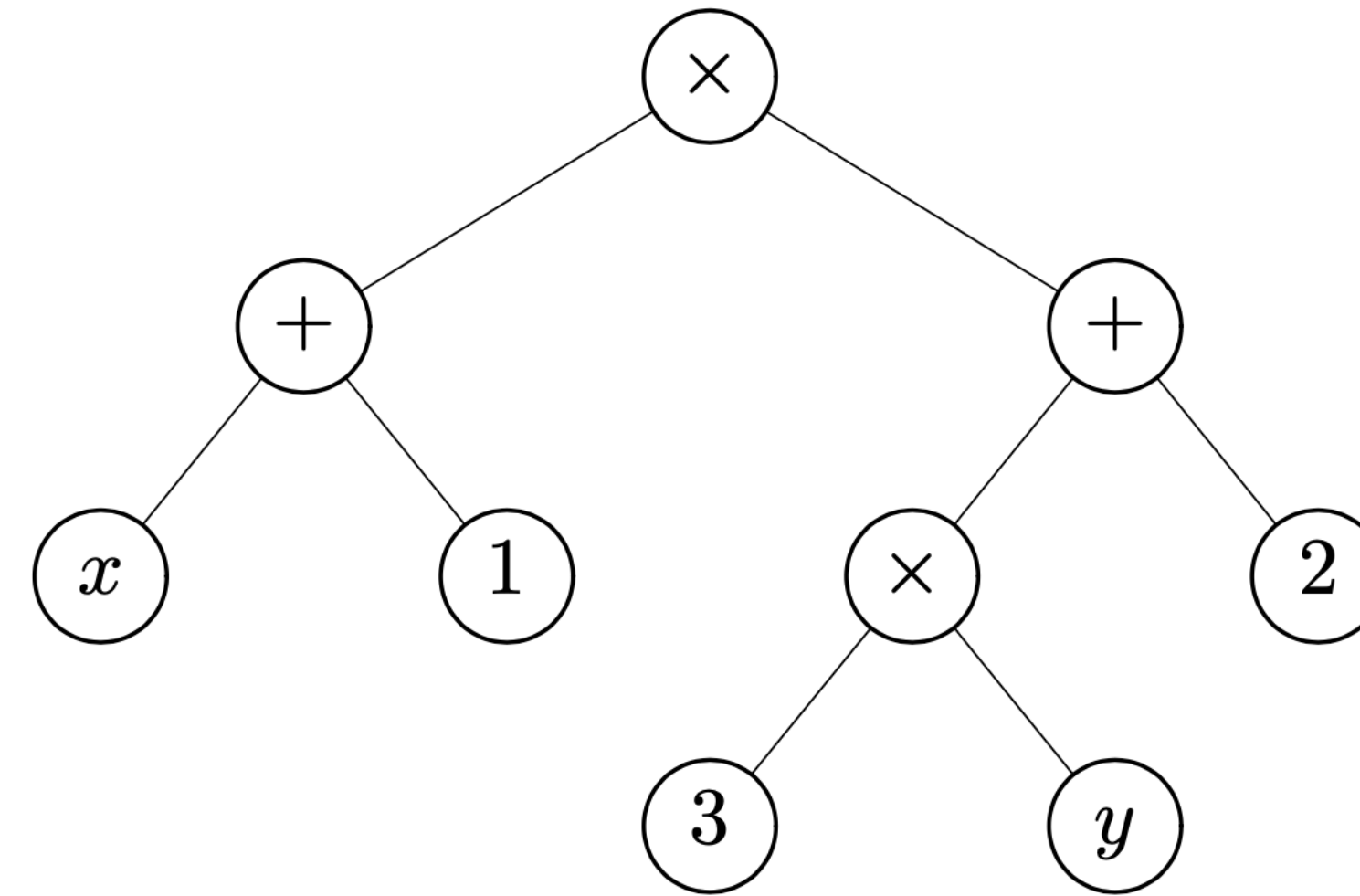
- passer d'une chaîne de caractères à un arbre (syntaxe abstraite) est plus difficile

$(x + 1) * (3 * y + 2)$

Analyse syntaxique



Belle impression



```
class Op_bi:
    def __init__(self, x, g, d) :
        self.val = x
        self.gauche = g
        self.droit = d
```

```
class Op_un:
    def __init__(self, x, a) :
        self.val = x
        self.fils = a
```

```
class CVar:
    def __init__(self, x) :
        self.val = x
```

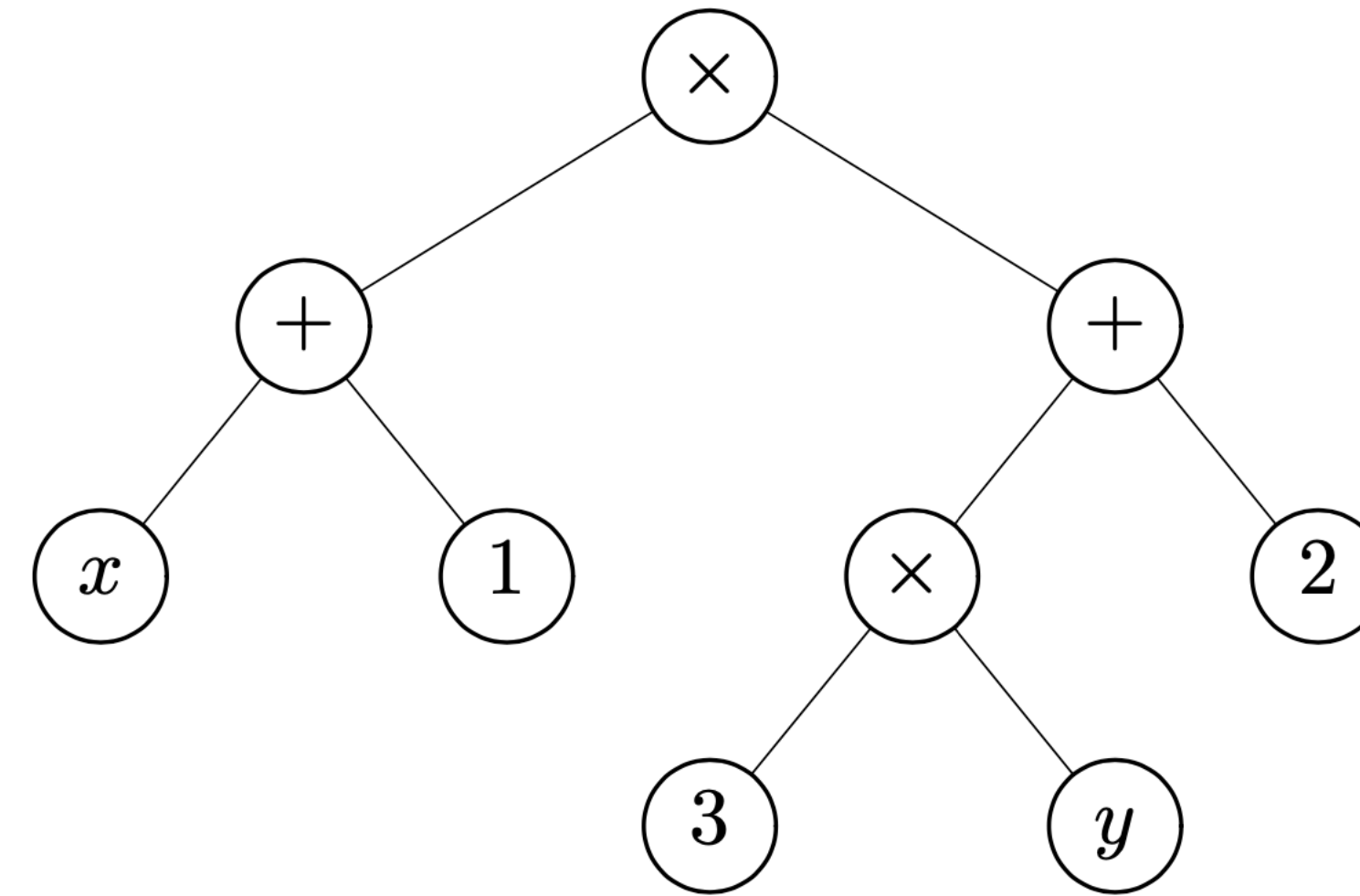
```
t = Op_bi('*', Op_bi('+', CVar('x'), CVar(1)),
            Op_bi('+', Op_bi('*', CVar(3), CVar('y')),
                    CVar(2)))
```


Arbres de syntaxe abstraite

- on peut évaluer sa valeur en donnant une valeur aux variables x et y
- on définit l'environnement par le dictionnaire:

```
e = {'x' : 20, 'y' : -20}
```

```
def eval (t, e) :  
    if isinstance (t, Op_bi) :  
        if t.val == '+' :  
            return eval (t.gauche, e) + eval (t.droit, e)  
        elif t.val == '*' :  
            return eval (t.gauche, e) * eval (t.droit, e)  
    elif isinstance (t, Op_un) :  
        return - eval (t.fils, e)  
    elif isinstance (t.val, int) :  
        return t.val  
    else :  
        return e[t.val]
```



```
t = Op_bi ('*', Op_bi ('+', CVar ('x'), CVar (1)),  
          Op_bi ('+', Op_bi ('*', CVar (3), CVar ('y')),  
                CVar (2)))
```

```
print (eval (t, e))
```

à faire

- retour sur les objets et les arbres
- analyses lexicales et syntaxiques
- modularité et programmation objet
- programmation graphique
- algorithmes géométriques
- calculs flottants et méthodes numériques
- programmation de plusieurs fils de calcul
- assertions et logique des programmes
- introduction à l'informatique théorique
- etc

vive l'informatique

et

la programmation !