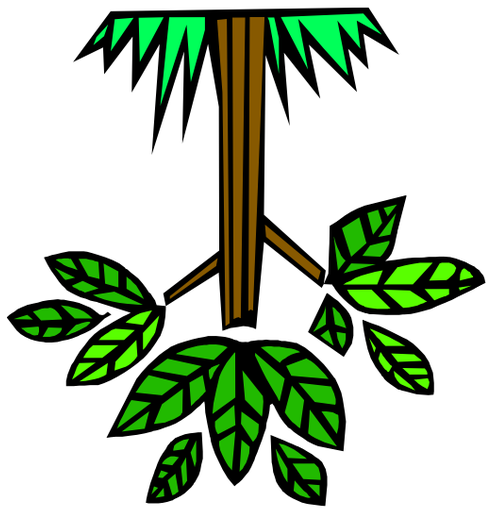
Université Nice Sophia Antipolis
Faculté des Sciences

Programmation avec Python - Printemps 2019
Etienne Lozes
L1-Sciences

Cours n°7

http://deptinfo.unice.fr/~elozes



Modules et Arbres

Type abstrait Pile

from  import  , 

1. Modules

from  import  , 

Les **modules** de Python

- Le **nom d'un fichier Python** se termine par `.py` et ne contient que des lettres minuscules, des chiffres et des soulignés. Aucun espace !

`essai2.py` `essai_tortue.py` ~~`Essai tortue.py`~~

- Un **module** est un fichier `xxxxxx.py` écrit en Python et contenant :
  - des définitions
  - des instructions (par exemple d'affichage)

- Un **module** peut être destiné :

- à être directement **exécuté**. Lorsqu'il est court et effectue une action ré-utilisable, on parle souvent d'un ***script***.	ou et	- à être **utilisé** par un autre module. Il exporte alors un certain nombre de fonctionnalités.

Un exemple de module utilitaire

fichier "tva.py"

Un module utilitaire

```
TVA = 19.6
COEF = 1 + TVA / 100
def prix_ttc(p) :
    global COEF
    return p * COEF
print('Module tva.py chargé !')
```

Mon programme

fichier "use_tva.py"

```
import tva
def bilan(x) :
    prix = tva.prix_ttc(x)
    print('Avec un taux de ',tva.TVA,end = '%, ')
    print('le prix ttc est ',prix)
bilan(26)
```

Exécution

```
Avec un taux de 19.6%, le prix ttc est 31.096
```

Portée des définitions d'un module

• Le module `tva.py` définit les variables `TVA`, `COEF` et la fonction `prix_ttc` . La portée de ces définitions est restreinte au module: elles ne sont pas visibles d'un autre module: dans use-tva, on peut définir une autre variable `COEF`, qui n'aura rien à voir avec la variable `COEF` de `tva.py`.

fichier "tva.py"

```
TVA = 19.6
COEF = 1 + TVA / 100
def prix_ttc(p) :
    global COEF
    return p * COEF
```

fichier "use-tva.py"

```
import tva
COEF = 0
print(tva.prix_ttc(1)) #->1.196
```

• L'existence d'espaces de noms (namespace) permet de protéger le programmeur de conflits entre des variables de même nom situées dans des modules distincts. La sécurité avant tout !

Que fait l'instruction 'import...' ?

- Utilisée dans le module `use-tva`, l'instruction `import tva` rend les définitions de `tva` accessibles depuis `use-tva`.
- Je peux alors accéder aux variables et fonctions définies dans le module `tva` en préfixant leur nom par le nom de leur module :

fichier "tva.py"

```
TVA = 19.6
COEF = 1 + TVA / 100
def prix_ttc(p) :
    global COEF
    return p * COEF
```

fichier "use-tva.py"

```
import tva
print(tva.TVA) # -> 19.6
tva.COEF = 0
print(tva.prix_ttc(1)) # -> 0
```

Que fait l'instruction 'from... import...' ?

• Utilisée dans un module M, l'instruction `from tva import TVA` introduit le mot TVA dans l'espace de noms du module M. Le mot TVA devient un nom de variable du module M :

```
>>> from tva import TVA
>>> TVA          # mais tva.TVA n'existe pas !!
19.6
```

*car le mot **tva** n'est pas importé !*

```
>>> from tva import TVA, prix_ttc
>>> (TVA, prix_ttc)
(19.6, <function prix_ttc at 0x10863c7a0>)
```

• Pour importer tout l'espace de noms du module `tva` :

`from tva import *`

N.B. En *général*, `import...` est *moins risqué* que `from... import...` car ce dernier peut introduire des conflits de noms !

Que fait l'instruction 'from... import... as...' ?

• Utilisée dans un module M, l'instruction `from tva import TAUX as T` introduit le mot `T` dans l'espace de noms du module courant. Le mot `T` devient un nom de variable du module M, remplaçant le nom `TAUX`.

• Imaginons que la variable `TVA` soit définie dans les modules `foo` et `bar`.

```
>>> from foo import TVA
>>> TVA
1
>>> from bar import TVA
>>> TVA
2
```

• Mieux vaut les nommer `foo.TVA` et `bar.TVA`, ou bien :

```
>>> from foo import TVA as FT
>>> from bar import TVA as BT
```

```
>>> (FT, BT)
(1, 2)
```

L'abstraction des données

- ***Maths***. Qu'est-ce qu'un *entier* ? Qu'est-ce qu'un *réel* ?

- Nous en avons une idée intuitive, nous pouvons même en nommer quelques uns : 25, 0.67, π... ***La plupart ne sont même pas calculables !***

- Nous ne savons pas vraiment *de quoi ils sont faits*. Heureusement nous connaissons leurs **propriétés** et leurs **opérations**. Et cela nous suffit !

- Les pros des maths s'occuperont de leur construction en temps utile...

Was sind und was sollen die Zahlen?
R. Dedekind, 1888

Quel rapport avec la programmation ?

• Nous avons programmé avec des objets élémentaires [nombres, booléens] et avec des objets composés [string, tuples, listes].

• Supposons que nous souhaitions programmer avec des ***matrices***. Notre langage ne les a pas prévues : il ne pouvait pas tout prévoir !

IDEE 1 : Nous allons **définir un nouveau type [abstrait] de donnée** ***matrice*** en utilisant des types de données déjà existants.

avec des structures, ou bien des listes, ou bien...

IDEE 2 : Nous allons nous empresser d'**oublier la manière dont ils ont été construits** pour nous focaliser sur leur utilisation à travers un ensemble de fonctions *abstraites*.

une boîte à outils "matrice"

Mise en oeuvre du type abstrait "matrice 2×2"

$$\begin{pmatrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

- Une **matrice 2×2** est la donnée de 4 nombres, organisés en lignes et colonnes. Il faut savoir construire une matrice et accéder à ses éléments.

Version 1 : implémentation par une liste

```
def matrice(a, b, c, d) :
  return [a, b, c, d]

def matrice_ref(M, li, co) : #Mi,j
   return M[2 * li + co]
```

```
>>> A = mat(1, 2, 3, 4)
>>> A
[1, 2, 3, 4]
> mat_ref(A, 1, 0)
3
```

Version 2 : implémentation par une liste de listes

```
def matrice(a, b, c, d) :
  return [[a, b], [c, d]]

def matrice_ref(M, li, co) :
   return M[li][co]
```

```
>>> A =matrice(1,2,3,4)
>>> A
[[1, 2], [3, 4]]
> matrice_ref(A, 1, 0)
3
```

Abstraction et modules

Conclusion : Il n'y a pas unicité de la représentation concrète d'une matrice abstraite.

L'important, c'est la matrice, pas sa représentation !

Mettons maintenant les fonctions de manipulation de matrice dans un module. En dehors du module, on n'a pas besoin de connaitre la représentation concrète des matrices, on peut en faire **abstraction**.

```
import matrice

def trace(M) :
  return matrice_ref(M, 0, 0) + matrice_ref(M, 1, 1)

def determinant(M) :
   return matrice_ref(M, 0, 0) * matrice_ref(M, 1, 1) \
        - matrice_ref(M, 1, 0) * matrice_ref(M, 0, 1)
```

un module qui fait abstraction de la représentation d'une matrice

```
def matrice(a, b, c, d) :
  return [a, b, c, d]

def matrice_ref(M, li, co) :
   return M[2 * li + co]
```

deux version possibles du module matrice.py

```
def matrice(a, b, c, d) :
  return [[a, b], [c, d]]

def matrice_ref(M, li, co) :
   return M[li][co]
```

• Quel que soit le choix de la structure concrète d'une matrice, on livre à l'utilisateur un ensemble de fonctions : ***la boîte à outils matrice 2×2*** :

• L'utilisateur n'a pas besoin de connaître le détail de l'implémentation. Une documentation suffit, avec la *complexité* des fonctions.

• Il va rédiger des algorithmes qui seront **indépendants de la représentation interne** d'une matrice.

• Si je change de boîte à outils sur les matrices, la fonction `det(M)` n'aura pas du tout à être modifiée !

• Si je travaille avec d'autres personnes sur un gros projet, je peux découper le problème en modules, et chacun travaille indépendamment sur son module.

• On dit que les modules permettent de définir un **type de données abstrait**.

Les **objets** sont un autre support à la modularisation et à l'abstraction du code.

Un autre exemple : les nombres rationnels

- Supposons que notre langage de programmation n'offre pas les **nombres rationnels exacts** comme 3/7, mais nous en avons besoin !

- Vite, un **Type Abstrait** !

- On représentera *mathématiquement* un rationnel ≠ 0 par une **fraction irréductible** unique p/q avec q > 0 et pgcd(p,q) = 1.

- On choisit de représenter *dans Python* un rationnel par un **couple** de deux entiers, numérateur et dénominateur, mêmes conditions.

- La boîte à outils ***nombres rationnels*** doit contenir le minimum de fonctions dépendant du choix du type de donnée concrète, ici la structure. Les autres fonctions sur les nombres rationnels n'auront pas à savoir qu'un rationnel est représenté par un couple!

fichier "rationnel.py"

```
def rationnel(p, q) : # le rationnel p/q avec p et q entiers
  if q == 0 :
     raise ValueError('dénominateur nul!')
  if q < 0 :
     (p, q) = (-p, -q)
  g = math.gcd(p, q)  # ici q est > 0
  return (p/g, q/g)

def numerateur(r) :
  return r[0]

def denominateur(r) :
  return r[1]

def repr(r) : # la représentation externe de r
   if r[0] == 0 : return '0'
   if r[1] == 1 : return str(r[0])
   return '{}/{}'.format(r[0], r[1])
```

- Exemple d'utilisation, l'algorithme abstrait d'**addition dans Q** :

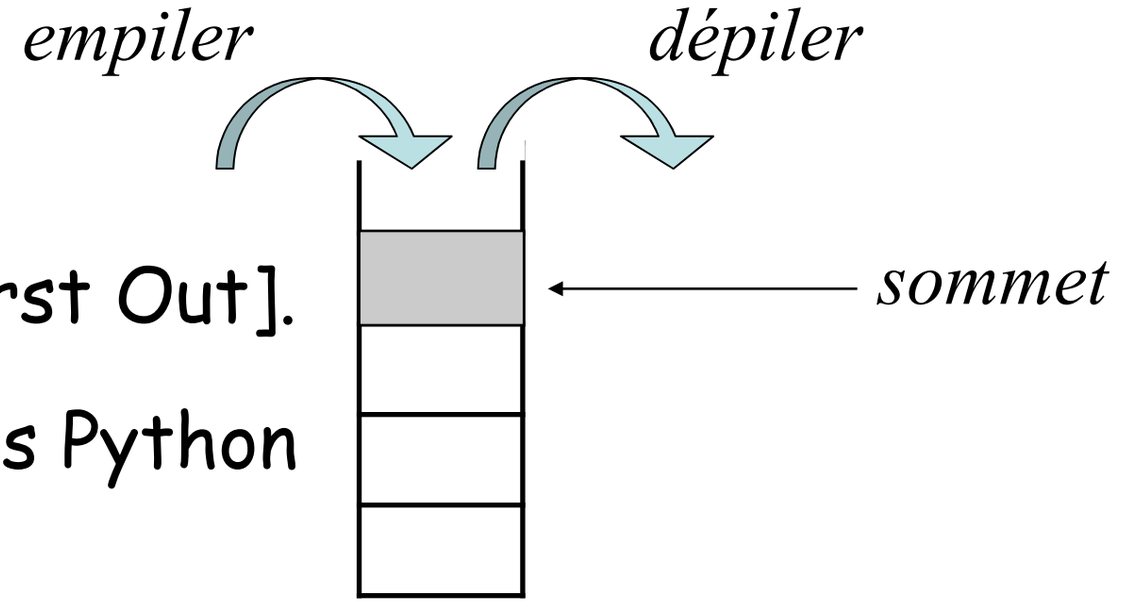
```
import rationnel

def add_rat(r1, r2) : # j'utilise le type abstrait!
  num = numerateur(r1) * denominateur(r2) \
      + numerateur(r2) * denominateur(r1)
  den = denominateur(r1) * denominateur(r2)
  return rationnel(num, den)
```

```
>>> r1 = rationnel(6, -4)
>>> r2 = rationnel(1, 3)
>>> r3 = rationnel(0, -5)
>>> print('r1 = {}'.format(repr(r1))
r1 = -3/2
>>> print('r2 = {}'.format(repr(r2))
r2 = 1/3
>>> print('r3 = {}'.format(repr(r3))
r3 = 0
>>> print('r1 + r2 = {}'.format(add_rat(r1, r2))
r1 + r2 = -7/6
```

obtention automatique d'une fraction irréductible !

Le type abstrait PILE

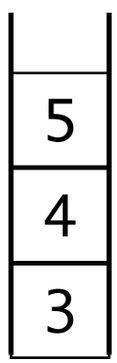


- Les **piles** sont des conteneurs fonctionnant en ordre LIFO [Last In, First Out].
- On choisit de représenter une pile dans Python par une liste.

```
def sommet(P) : return P[-1]
def empiler(P, v) : P.append(v)
def dépiler(P) : return P.pop()
def pile_vide() : return []
def est_vide(p) : return p == []
```

Les fonctions `empiler(x, P)` et `dépiler(P)` modifient la pile `P`.

```
>>> P = pile_vide()
>>> empiler(P,3) ; empiler(P, 4) ; empiler(P, 5)
>>> sommet(P)
5
> dépiler(P)
5
> sommet(P)
4
```



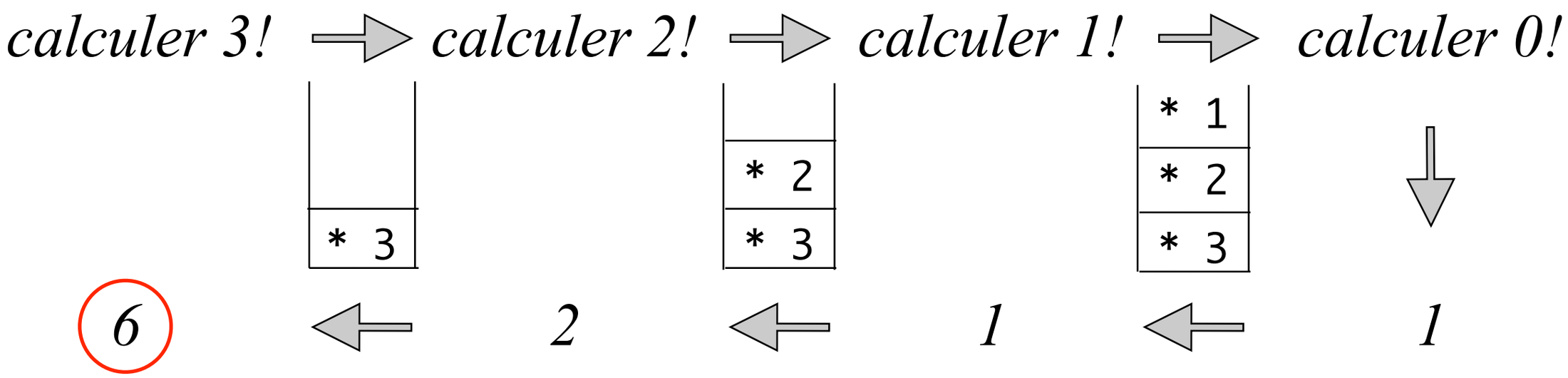
A quoi sert une PILE ?

• En électricité, à fournir de l'énergie. En cuisine, à ranger des assiettes. En programmation, à **stocker des données intermédiaires** qu'il faudra utiliser plus tard pour **revenir en arrière**.

• Par exemple, pour exécuter une ***fonction récursive***, une pile implicite [non visible au programmeur] est utilisée par le système.

```
fac(n) = return n * fac(n-1)
```

Pour calculer n! , je calcule (n-1)! mais auparavant je mets la multiplication par n en attente dans une pile...



Premier exemple : une animation processing réversible

Reprenons le dessin à main levé à la souris.
On veut rajouter une fonction « annuler » à notre programme.
Utilisons une pile!

La pile va contenir tous les traits qui ont été tracés:

- si la souris est pressée et déplacée, on empile un nouveau trait
- si une touche est pressée, on annule le dernier trait: on le dépile!

Enfin, pour décrire un trait, il suffit de connaitre ses extrémités.
On va donc représenter un trait par un couple de points.

Le monde est un triplet **(P, px, py)** : P est la pile des traits tracés, et (px,py) sont les coordonnées de la souris à sa dernière position enregistrée.

```
def suivant(m) :
    # ON MET A JOUR LA POSITION DE LA SOURIS
    (P, px, py) = m
    return (P, mouseX, mouseY)
```

```
def dessiner(m) :
    (P, px, py) = m
    if mousePressed:
        # si la souris a été pressée, on ajoute
        # un trait à la pile
        empiler(P, ((px,py), (mouseX, mouseY)))
    # on efface tout et on redessine tous les
    # traits présents dans la pile P
    background(255, 255, 255)
    for ((x1,y1),(x2,y2)) in P :
        line(x1,y1,x2,y2)
```

```
def setup() :
    global m
    size(LARG, HAUT)
    stroke(0)
    frameRate(28)
    # le monde initial
    m = (pile_vide(), 0, 0)
```

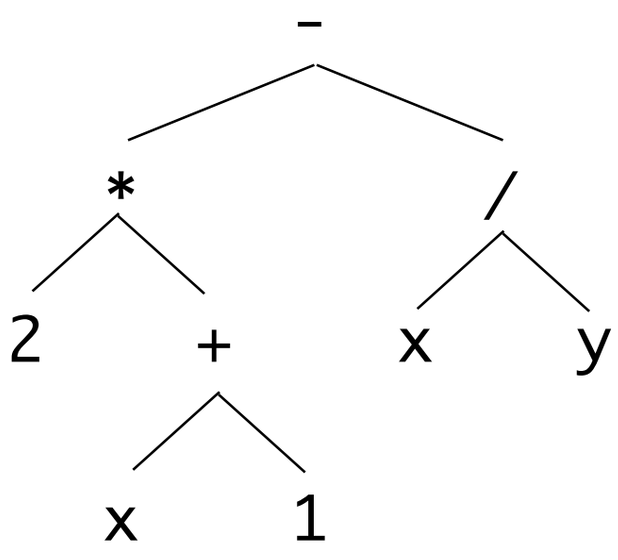
```
def keyPressed() :
    global m
     # si un bouton est pressé, on annule le
     # dernier trait = on le dépile
    (P, px, py) = m
    depiler(P)
```

```
def draw() :
    global m
    dessiner(m)
    m = suivant(m)
```

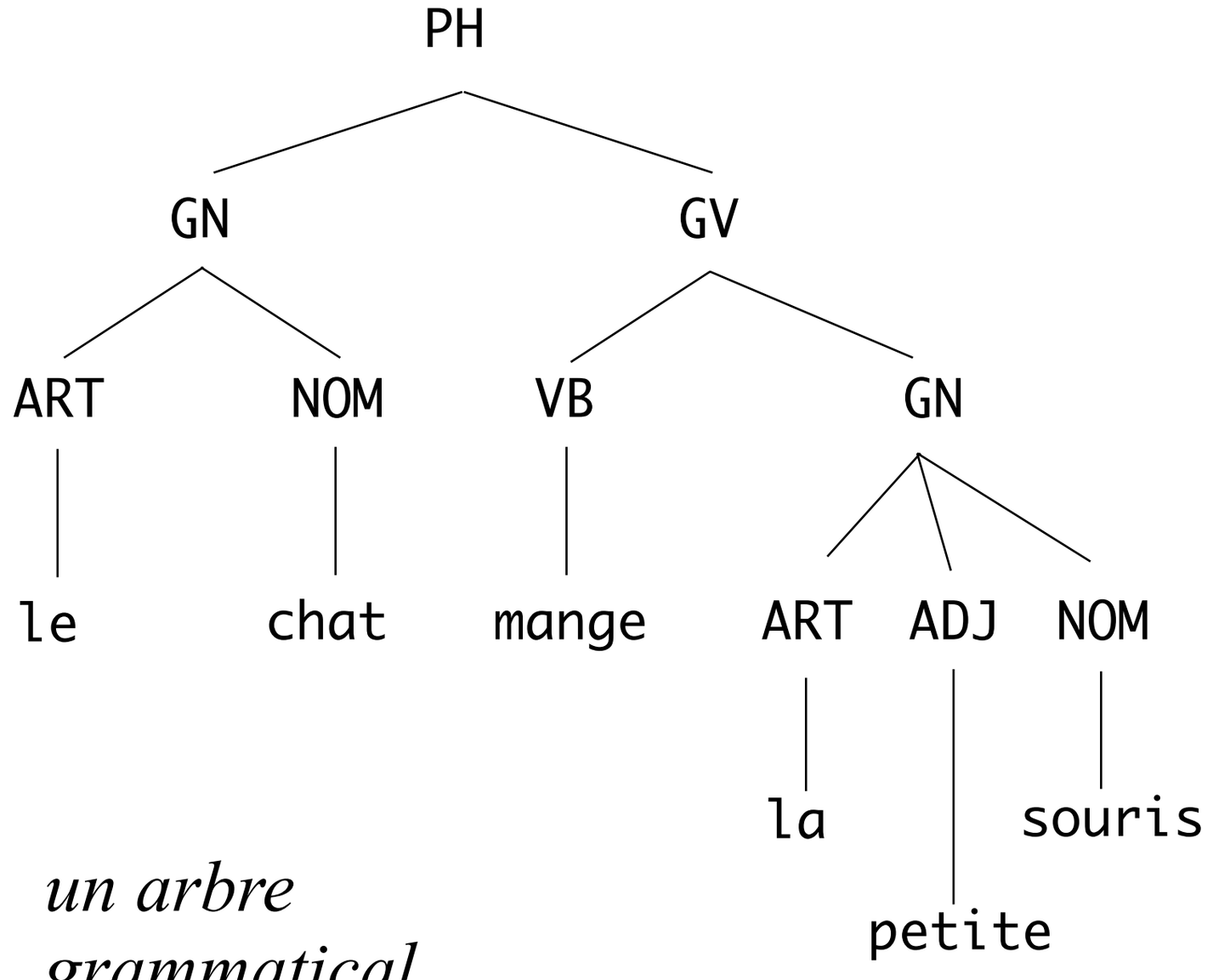
2. Arbres binaires d'expression

Divers types d'arbres

- Il y a plusieurs types d'arbres possibles, strictement binaires ou ayant un nombre variable de fils :



un arbre binaire d'expression



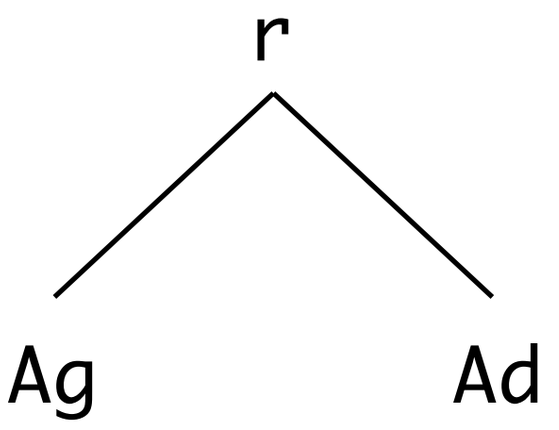
un arbre grammatical (1-2-3)

Les arbres binaires d'expressions

- Nous allons dans un premier temps centrer notre étude sur les **arbres binaires d'expressions algébriques** : analyser, transformer, compiler !

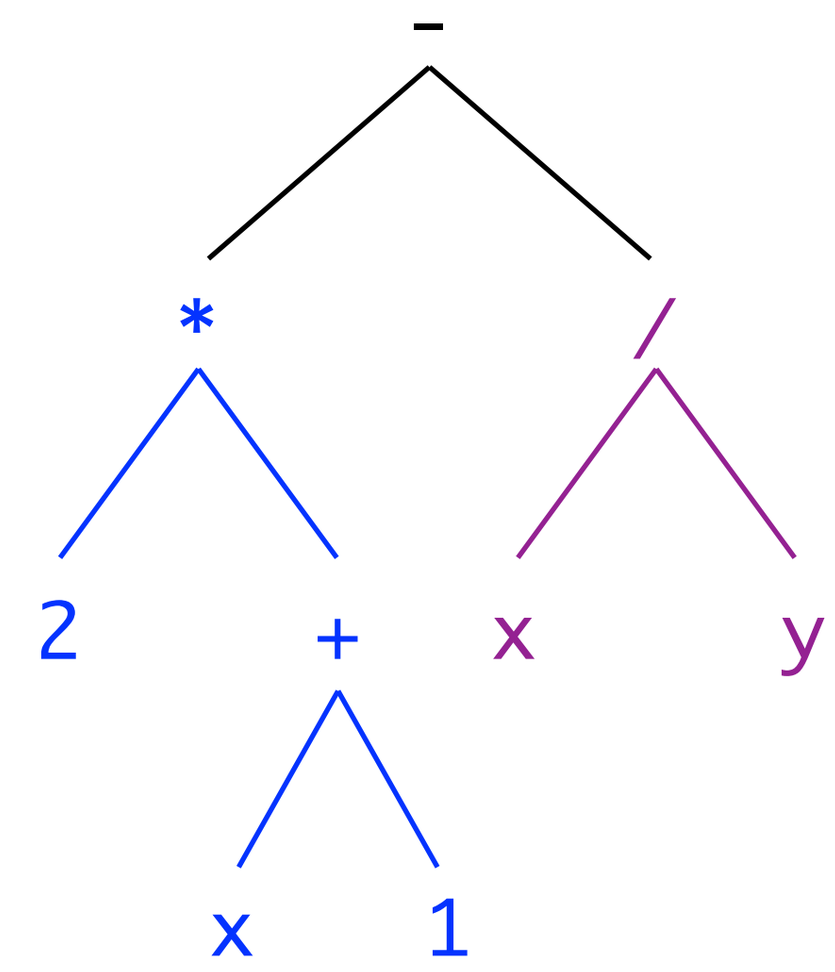
2 est un **arbre** [une *feuille*]

x est un **arbre** [une *feuille*]



est un **arbre** [un *noeud*] si :

- r est un opérateur
- Ag est un arbre
- Ad est un arbre



2*(x+1)-x/y

Le type abstrait "arbre binaire d'expression"

- Un **arbre binaire d'expression** sera représenté :
  - si c'est une ***feuille***, directement par cette feuille.
  - si c'est un ***noeud***, par une liste à 3 éléments `[r, Ag, Ad]`

```
def arbre(r, Ag, Ad) :
  return [r, Ag, Ad]
```

```
def est_feuille(obj) :
  return type(obj) == int or type(obj) == str
```

```
def est_operateur(obj) :
  return obj in ['+', '*', '-', '/']
```

- **NB** : Il n'y a ***pas d'arbre vide*** dans cette théorie !

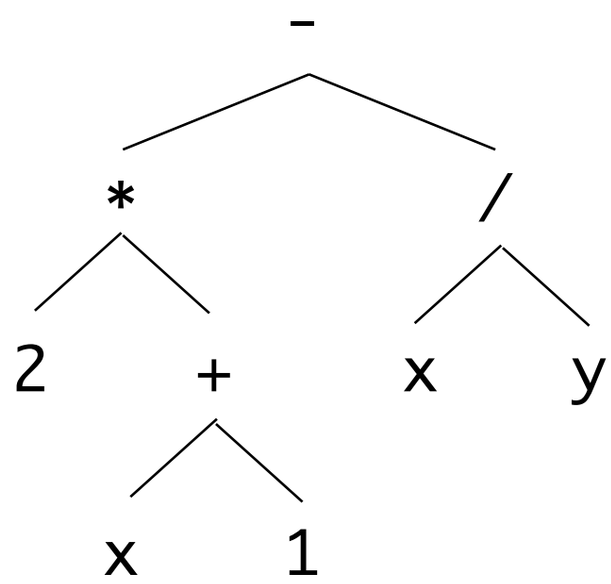
- Les trois accesseurs suivent la grammaire :

```
def racine(A) :  # A est un noeud
  if est_feuille(A) :
    raise ValueError("une feuille n'a pas de racine")
  return A[0]
```

```
def fg(A) :  # A est un noeud
  if est_feuille(A) :
    raise ValueError("une feuille n'a pas de fils gauche")
  return A[1]
```

```
def fd(A) :  # A est un noeud
  if est_feuille(A) :
    raise ValueError("une feuille n'a pas de fils droit")
  return A[2]
```

Construction d'un arbre



- Pour construire un arbre, on peut :

  - soit passer proprement par le constructeur du type abstrait :

```
AT = arbre('-', arbre('*', 2, 'x'), arbre('/', 'x', 'y'))
    # un arbre pour les tests
```

  - soit *outrepasser le type abstrait* et utiliser directement une liste ; c'est mal, et on le fera uniquement pour les tests :

```
AT = ['-',  ['*',  2, 'x'], ['/', 'x', 'y']]
```

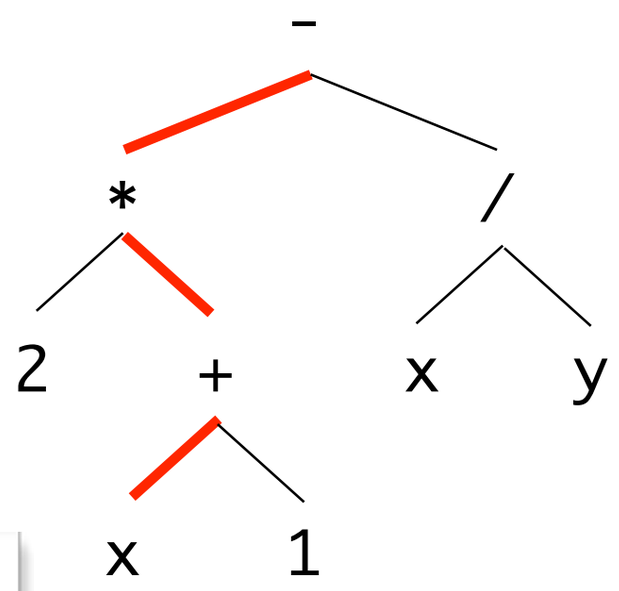
- Quoiqu'il en soit :

```
>>> AT
['-',  ['*',  2, 'x'], ['/', 'x', 'y']]
>>> fg(AT)
['*', 2, 'x']
```

```
> fd(fg(AT))
['+', 'x', 1]
> racine(fd(AT))
'/'
```

Quelques algorithmes de base

Hauteur d'un arbre



- C'est la longueur d'un chemin de longueur maximum reliant la racine à une feuille :

```
def hauteur(A) :
  if est_feuille(A) :
    return 0
  return 1 + max(hauteur(fg(A)), hauteur(fd(A)))
```

```
>>> hauteur(AT)
3
```

Notez la récurrence double !

N.B. Un arbre de hauteur h contient au plus 2^h feuilles. S'il en contient exactement 2^h on dit que c'est un **arbre binaire complet**. Inversement, s'il contient n feuilles, on s'attend à ce qu'il ait une hauteur telle que $n=2^h$, d'où $h=\log_2 n$ en moyenne...

Présence d'une feuille

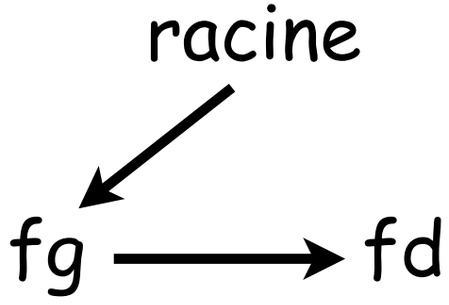
- La **présence d'une feuille** donnée x dans un arbre A.

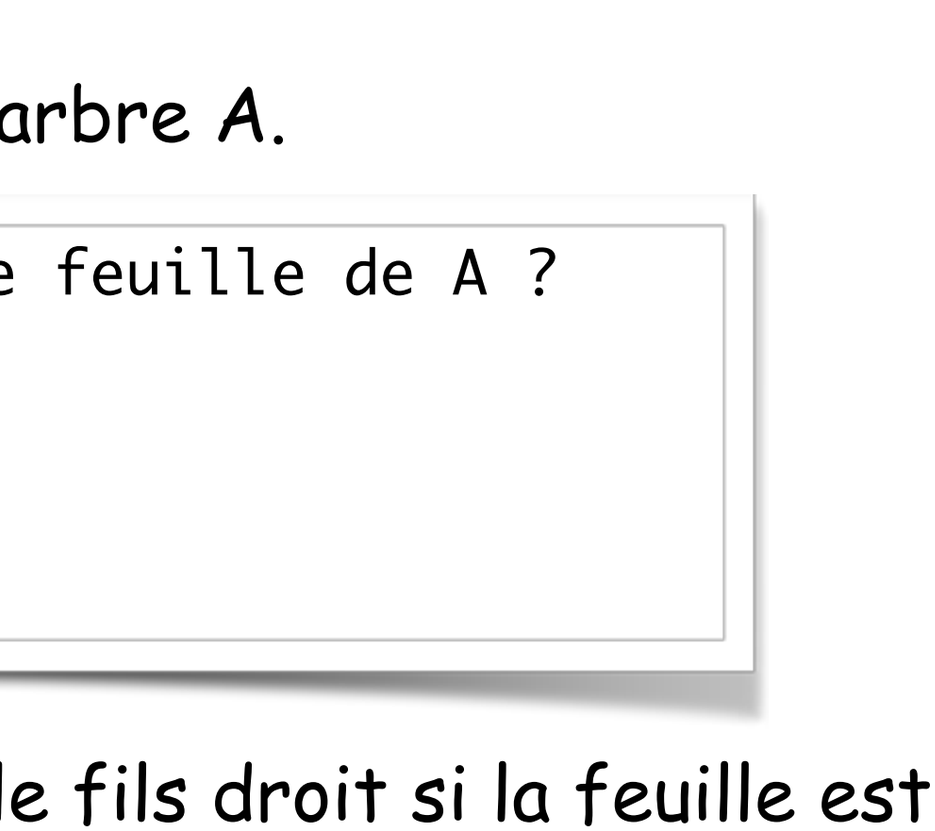
```
def est_feuille_de(f, A) :  # f est une feuille de A ?
  if est_feuille(A) :
      return f == A
  return est_feuille_de(f, fg(A)) or \
         est_feuille_de(f, fd(A))
```

- Notez que le or paresseux évite d'explorer le fils droit si la feuille est trouvée à gauche !

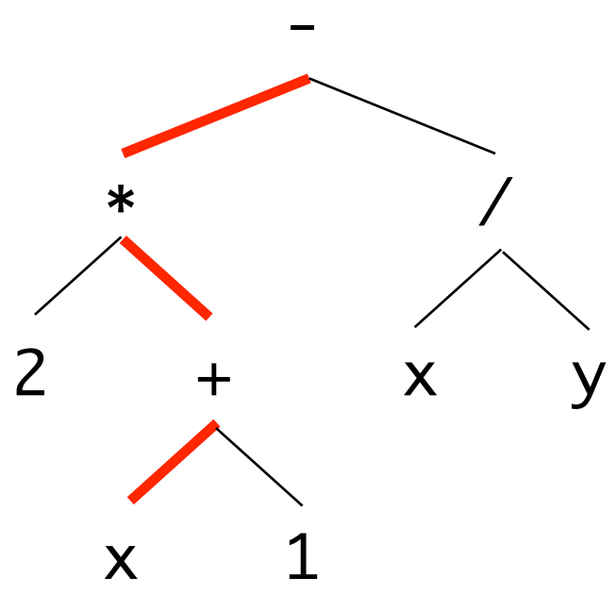
DEFINITION : Un parcours est **en profondeur** si l'un des fils est complètement exploré avant d'entamer l'exploration de l'autre !

- Le parcours ci-contre est donc un *parcours en profondeur préfixe*.





Feuillage d'un arbre



- Calculer le **feuillage d'un arbre** revient à produire la liste *plate* de ses feuilles dans un parcours en profondeur préfixe :

```
def feuillage(A) :
  if est_feuille(A) :
      return [A]
  return feuillage(fg(A)) + feuillage(fd(A))
```

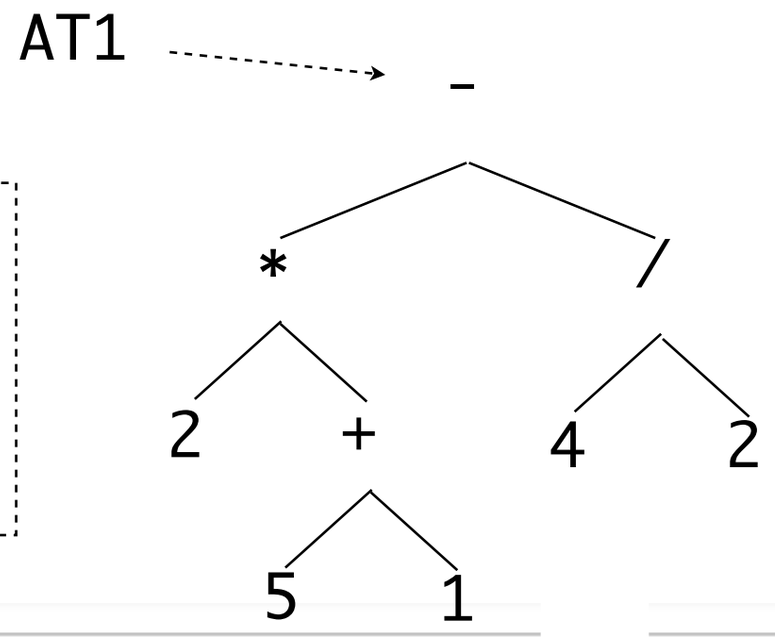
```
>>> feuillage(AT)
[2, 'x', 1, 'x', 'y']
```

- Variante : le **nombre de feuilles**.

```
def nombre_de_feuilles(A) :
  if est_feuille(A) :
      return 1
  return nombre_de_feuilles(fg(A)) + nombre_de_feuilles(fd(A))
```

```
>>> nombre_de_feuilles(AT)
5
```

Valeur d'un arbre arithmétique

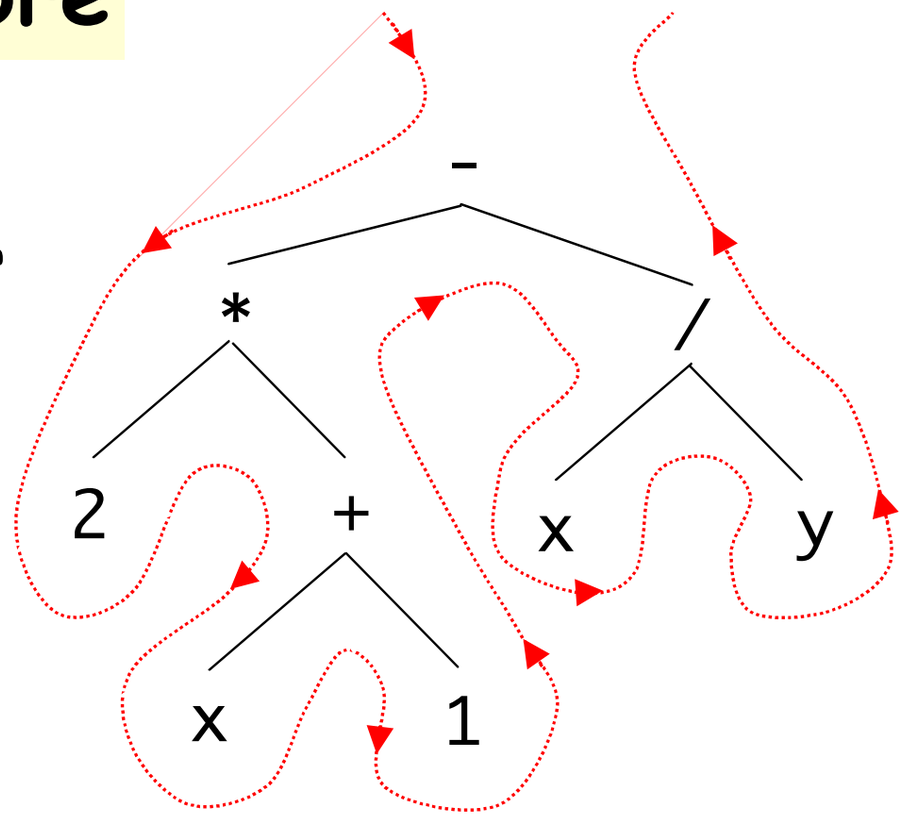


DEFINITION : Un arbre sera dit **arithmétique** si toutes ses feuilles sont des **constantes**. Il est **algébrique** si au moins une feuille est une variable.

```
def valeur(A) :  # l'arbre A est arithmétique
  if est_feuille(A) :
      return A
  r = racine(A)
  vg = valeur(fg(A))
  vd = valeur(fd(A))
  if r ==  '+' : return vg + vd
  if r ==  '-' : return vg - vd
  if r ==  '*' : return vg * vd
  if r ==  '/' : return vg / vd
  raise ValueError('valeur : opérateur inconnu')
```

```
>>> valeur(AT1)
10
```

Parcours profondeur préfixe d'un arbre



- Etant donné un arbre A, on cherche à obtenir la liste de tous les éléments rencontrés lors d'un **parcours en profondeur préfixe** :

```
>>> pp_prefixe(['-', ['*', 2, ['+', 'x', 1]] ['/', 'x', 'y']])
['-', '*', 2, '+', 'x', 1, '/', 'x', 'y']
```

- En quelque sorte, on a *enlevé les crochets intérieurs* !

```
def pp_prefixe(A) :  # Arbre → Liste
  if est_feuille(A) :
    return [A]
  return [racine(A)] + pp_prefixe(fg(A)) + pp_prefixe(fd(A))
```

- Saurait-on remettre les crochets intérieurs, i.e. programmer la fonction réciproque ?

cf TD (si le temps le permet)!

Autre application des piles : parcours itératif d'un arbre

• Les arbres binaires formant un type de donnée récursif, il est naturel de les programmer récursivement !

• Mais il peut être intéressant de produire un algorithme itératif, avec une boucle while.

• La présence de deux fils introduit des mises en attente. Munissons-nous donc d'**une pile en paramètre** et empilons-y les sous-arbres qu'il restera à visiter, pour pouvoir y revenir plus tard !

• Exemple : soit à calculer le **nombre de feuilles** d'un arbre A.

STRATEGIE DU PARCOURS EN PROFONDEUR ITERATIF :

- si je suis sur un noeud, j'empile [un pointeur sur] le fils droit et je vais visiter le fils gauche.

- si je suis sur une feuille, je regarde s'il reste un arbre à visiter au sommet de la pile.

```
def nb_feuilles(A) : # le nombre de feuilles
  P = pile_vide() ; empiler(P, A)
  res = 0
  while not est_vide(P) :  # il reste des sous-arbres à visiter
     A = depiler(P)
     print('A={} P={} res={}'.format(A, P, res)) # pour le debug...
     if est_feuille(A) :
        res = res + 1
     else :
        empiler(P, fd(A)) ; empiler(P, fg(A))
  return res
```

```
>>> nb_feuilles(['-', ['*', 2, ['+', 'x', 1]], ['/', 'x', 'y']])
A=['-', ['*', 2, ['+', 'x', 1]], ['/', 'x', 'y']] P=[] res=0
A=['*', 2, ['+', 'x', 1]] P=[['/', 'x', 'y']] res=0
A=2 P=[['+', 'x', 1] ['/', 'x', 'y']] res=0
A=['+', 'x', 1] P=[['/', 'x', 'y']] res=1
A=x P=[1, ['/', 'x', 'y']] res=1
A=1 P=[['/', 'x', 'y']] res=2
A=['/', 'x', 'y'] P=[] res=3
A='x' P=['y'] res=3
A='y' P=[] res=4
==> 5
```