Bases de l'informatique 2

Cours bonus. Types récursifs en Python

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 1 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

- La pile est un type de structure, comme le tuple ou la liste, très utilisé en informatique.

Le type abstrait Pile

- La pile est un type de structure, comme le tuple ou la liste, très utilisé en informatique.

- Fonctionnement :
  - Au début la pile est vide

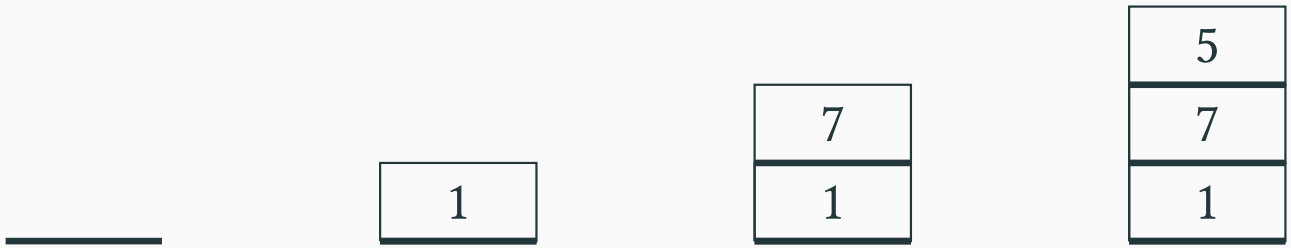
- La pile est un type de structure, comme le tuple ou la liste, très utilisé en informatique.

- Fonctionnement :
  - Au début la pile est vide
  - On empile 1

- La pile est un type de structure, comme le tuple ou la liste, très utilisé en informatique.

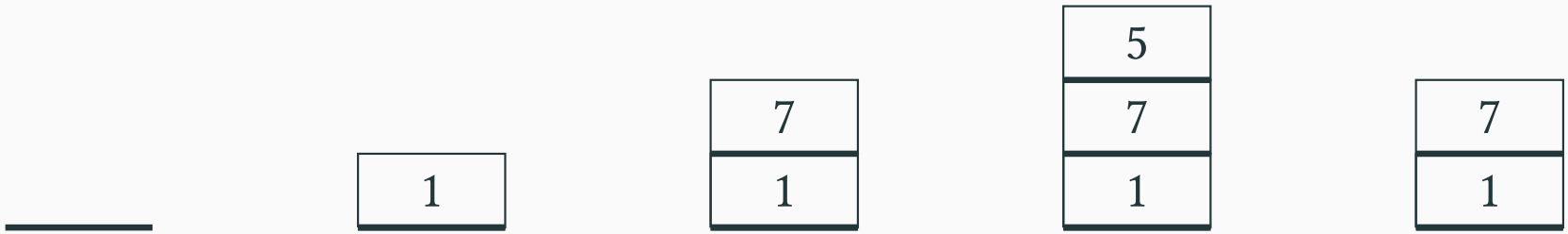
- Fonctionnement :
  - Au début la pile est vide
  - On empile 1
  - On empile 7

- La pile est un type de structure, comme le tuple ou la liste, très utilisé en informatique.



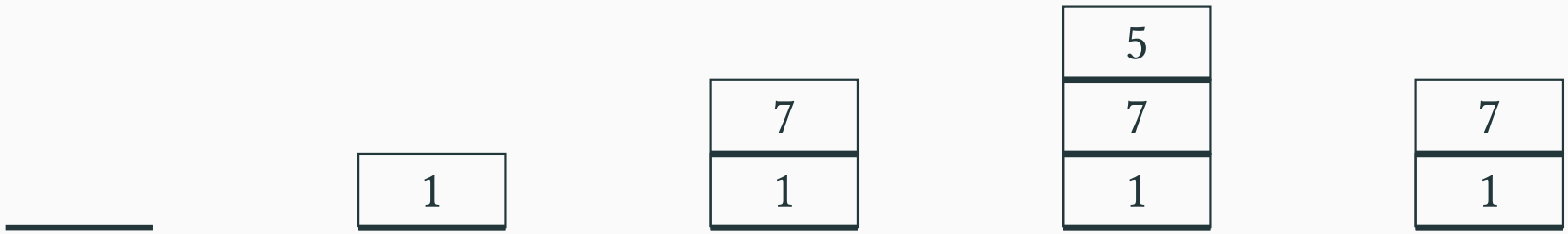
- Fonctionnement :
  - Au début la pile est vide
  - On empile 1
  - On empile 7
  - On empile 5

- La pile est un type de structure, comme le tuple ou la liste, très utilisé en informatique.



- Fonctionnement :
  - Au début la pile est vide
  - On empile 1
  - On empile 7
  - On empile 5
  - On dépile 5

- La pile est un type de structure, comme le tuple ou la liste, très utilisé en informatique.



- Fonctionnement :
  - Au début la pile est vide
  - On empile 1
  - On empile 7
  - On empile 5
  - On dépile 5
  - À la fin le sommet vaut 7

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>>
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>>
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
>>>
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
>>> pile.empile(P,3)
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
>>> pile.empile(P,3)
>>>
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
>>> pile.empile(P,3)
>>> pile.sommet(P)
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
>>> pile.empile(P,3)
>>> pile.sommet(P)
3
>>>
```

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
>>> pile.empile(P,3)
>>> pile.sommet(P)
3
>>> pile.dépile(P)
```

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
>>> pile.empile(P,3)
>>> pile.sommet(P)
3
>>> pile.dépile(P)
3
>>>
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
>>> pile.empile(P,3)
>>> pile.sommet(P)
3
>>> pile.dépile(P)
3
>>> pile.sommet(P)
```

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

```
>>> import pile
>>> P = pile.nouvelle_pile()
>>> pile.empile(P,3)
>>> pile.sommet(P)
3
>>> pile.dépile(P)
3
>>> pile.sommet(P)
Traceback (most recent call last):
  File "<console>", line 1, in
<module>
  File "pile.py", line 13, in
sommet
    if L == []: raise PileErreur
                ^^^^^^^^^^^^^^^^

ValueError:  Pile vide
```

À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.

À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.
- Typiquement pour le calcul de la factorielle.
  - Il faut calculer `fact(n-1)` avant de faire le `* n`
  - On met les calculs « `* n` » dans la pile.
  - et dès qu'on tombe sur `0!` on dépile et applique les calculs

SCRIPT

```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n
```

À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.
- Typiquement pour le calcul de la factorielle.
  - Il faut calculer `fact(n-1)` avant de faire le `* n`
  - On met les calculs « `* n` » dans la pile.
  - et dès qu'on tombe sur `0!` on dépile et applique les calculs

SCRIPT

```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n
```

3 !

À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.
- Typiquement pour le calcul de la factorielle.
  - Il faut calculer `fact(n-1)` avant de faire le `* n`
  - On met les calculs « `* n` » dans la pile.
  - et dès qu'on tombe sur `0!` on dépile et applique les calculs

SCRIPT

```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n
```

À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.
- Typiquement pour le calcul de la factorielle.
  - Il faut calculer `fact(n-1)` avant de faire le `* n`
  - On met les calculs « `* n` » dans la pile.
  - et dès qu'on tombe sur `0!` on dépile et applique les calculs

SCRIPT

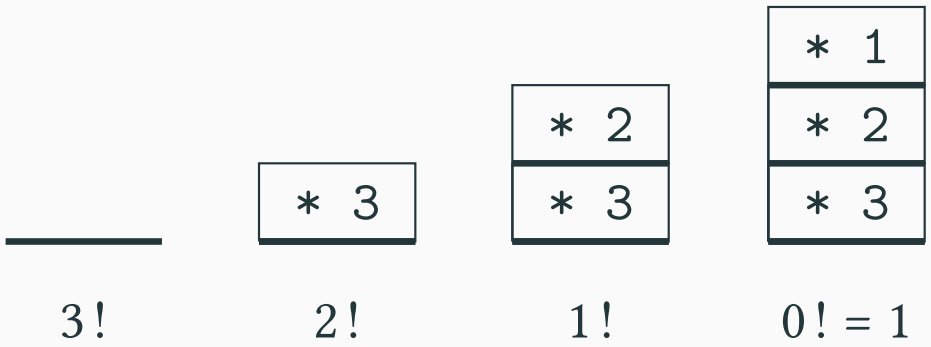
```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n
```

À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.
- Typiquement pour le calcul de la factorielle.
  - Il faut calculer `fact(n-1)` avant de faire le `* n`
  - On met les calculs « `* n` » dans la pile.
  - et dès qu'on tombe sur `0!` on dépile et applique les calculs

SCRIPT

```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n
```

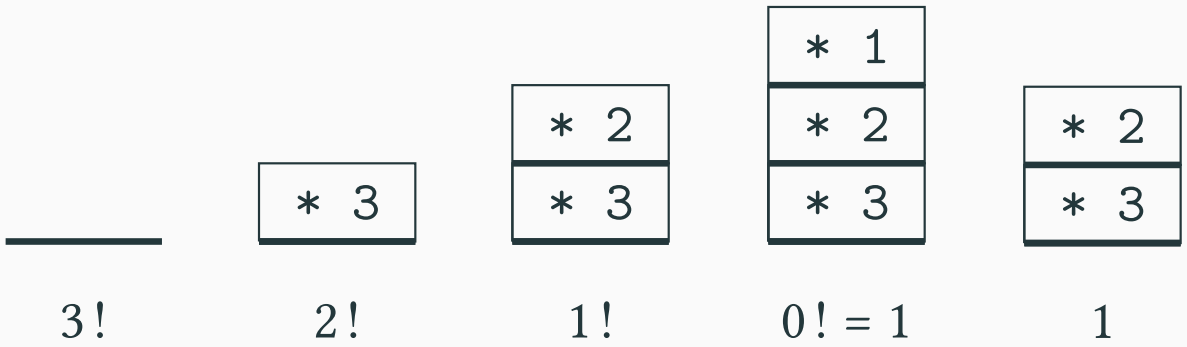


À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.
- Typiquement pour le calcul de la factorielle.
  - Il faut calculer `fact(n-1)` avant de faire le `* n`
  - On met les calculs « `* n` » dans la pile.
  - et dès qu'on tombe sur `0!` on dépile et applique les calculs

SCRIPT

```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n
```

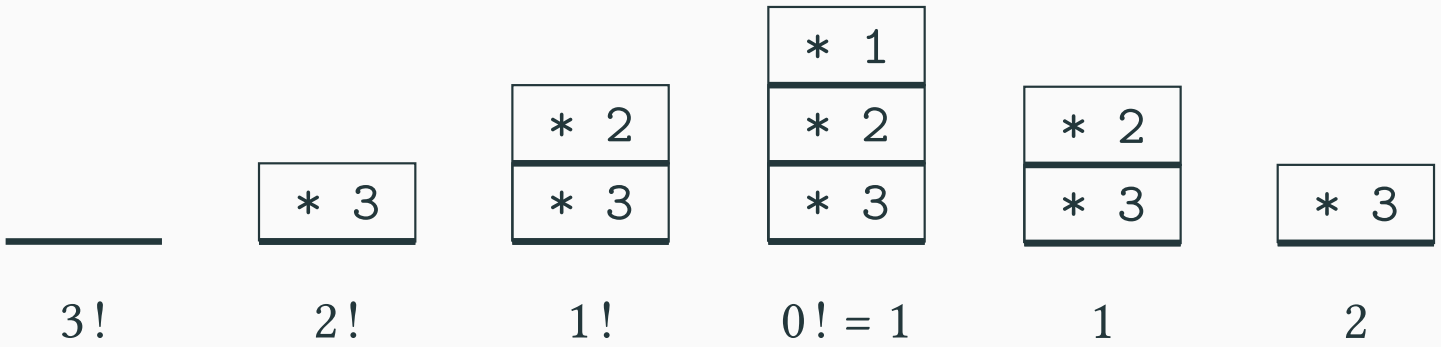


À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.
- Typiquement pour le calcul de la factorielle.
  - Il faut calculer `fact(n-1)` avant de faire le `* n`
  - On met les calculs « `* n` » dans la pile.
  - et dès qu'on tombe sur `0!` on dépile et applique les calculs

SCRIPT

```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n
```

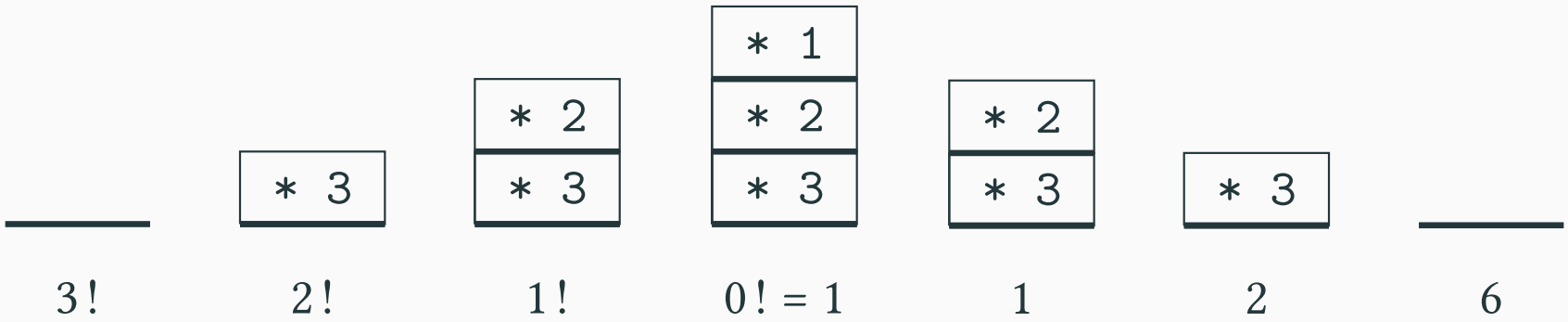


À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.
- Typiquement pour le calcul de la factorielle.
  - Il faut calculer `fact(n-1)` avant de faire le `* n`
  - On met les calculs « `* n` » dans la pile.
  - et dès qu'on tombe sur `0!` on dépile et applique les calculs

SCRIPT

```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n
```



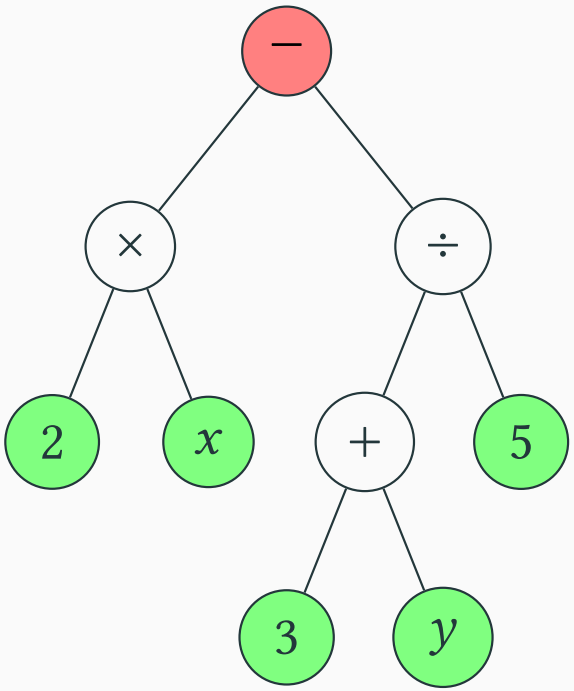
- L'appel classique fonctionne aussi avec des piles : BI1 CM6 partie v

Sommaire

- Un arbre (graphe acyclique) est un objet fondamental en informatique.
  - Un arbre a une racine (en rouge)
  - Un arbre a des nœuds (avec des enfants) ; la racine est un nœud
  - Un arbre a des feuilles (sans enfant) (en vert)

- Un arbre (graphe acyclique) est un objet fondamental en informatique.
  - Un arbre a une racine (en rouge)
  - Un arbre a des nœuds (avec des enfants) ; la racine est un nœud
  - Un arbre a des feuilles (sans enfant) (en vert)

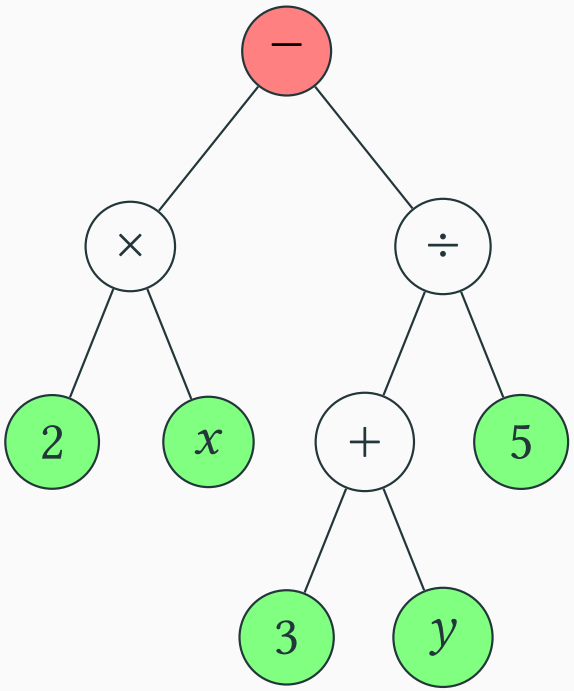
Arbre binaire d'expression



1 racine, 4 nœuds, 5 feuilles

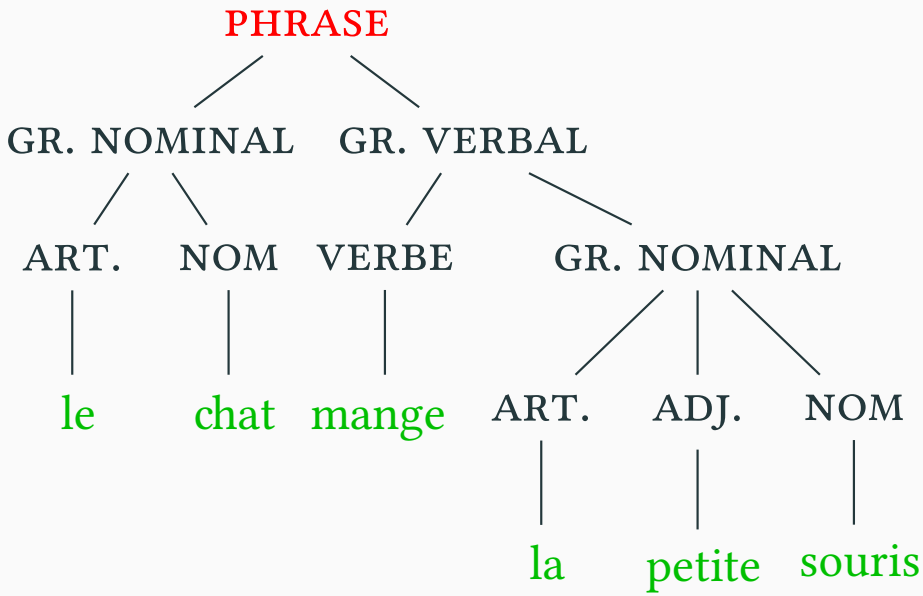
- Un arbre (graphe acyclique) est un objet fondamental en informatique.
  - Un arbre a une racine (en rouge)
  - Un arbre a des nœuds (avec des enfants) ; la racine est un nœud
  - Un arbre a des feuilles (sans enfant) (en vert)

Arbre binaire d'expression



1 racine, 4 nœuds, 5 feuilles

Un arbre grammatical



1 racine, 10 nœuds, 6 feuilles

Les arbres binaires d'expression : principe

- Il y a deux types d'arbres :

Les arbres binaires d'expression : principe

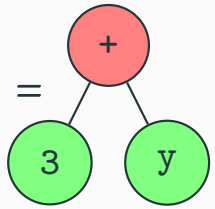
- Il y a deux types d'arbres :
  - Les arbres simples correspondant à des feuilles

Les arbres binaires d'expression : principe

- Il y a deux types d'arbres :
  - Les arbres simples correspondant à des feuilles
  - Les arbres composés constitués d'une racine et deux sous-arbres

Les arbres binaires d'expression : principe

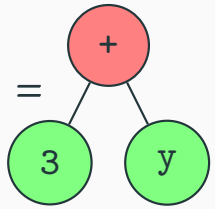
- Il y a deux types d'arbres :
  - Les arbres simples correspondant à des feuilles
  - Les arbres composés constitués d'une racine et deux sous-arbres
- On choisit une opération r, un arbre composé A_1, un arbre simple A_2 :

$r = /$ $\quad A_1 =$ 

$\quad A_2 =$ 

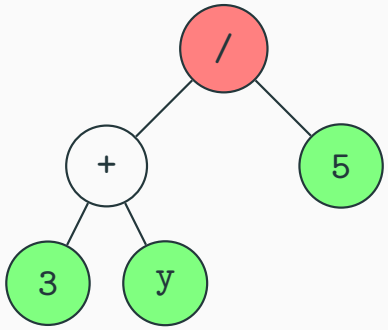
Les arbres binaires d'expression : principe

- Il y a deux types d'arbres :
  - Les arbres simples correspondant à des feuilles
  - Les arbres composés constitués d'une racine et deux sous-arbres
- On choisit une opération r, un arbre composé A_1, un arbre simple A_2 :

$r = /$ $\quad A_1 =$ 

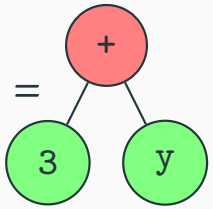
$\quad A_2 =$ 

- On peut construire un nouvel arbre composé :



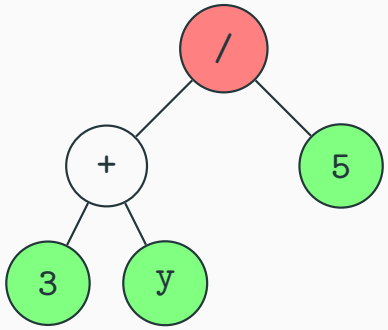
Les arbres binaires d'expression : principe

- Il y a deux types d'arbres :
  - Les arbres simples correspondant à des feuilles
  - Les arbres composés constitués d'une racine et deux sous-arbres
- On choisit une opération r, un arbre composé A_1, un arbre simple A_2 :

$r = /$ $\quad A_1 =$ 

$\quad A_2 =$ 

- On peut construire un nouvel arbre composé :



- Chaque arbre correspond à une expression : ici l'expression est `(3+y)/5`

Les arbres binaires d'expression : définitions

- On définit un arbre binaire d'expression (ABE) par récurrence :

Les arbres binaires d'expression : définitions

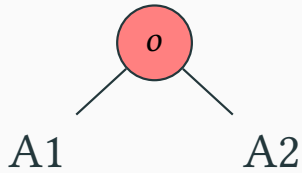
- On définit un arbre binaire d'expression (ABE) par récurrence :
- Cas de base : une feuille contenant un nombre ou une lettre est un ABE.

Les arbres binaires d'expression : définitions

- On définit un arbre binaire d'expression (ABE) par récurrence :
- Cas de base : une feuille contenant un nombre ou une lettre est un ABE.
  - L'expression correspondante est le contenu de la feuille.

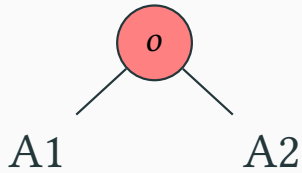
Les arbres binaires d'expression : définitions

- On définit un arbre binaire d'expression (ABE) par récurrence :
- Cas de base : une feuille contenant un nombre ou une lettre est un ABE.
  - L'expression correspondante est le contenu de la feuille.
- Cas composé : L'arbre suivant est une ABE si et seulement si :
  - A_1 et A_2 sont des ABE
  - o une operation (`+`, `-`, `*`, `/`)
  - L'expression associée s'obtient en appliquant o aux expressions associées à A_1 et A_2.

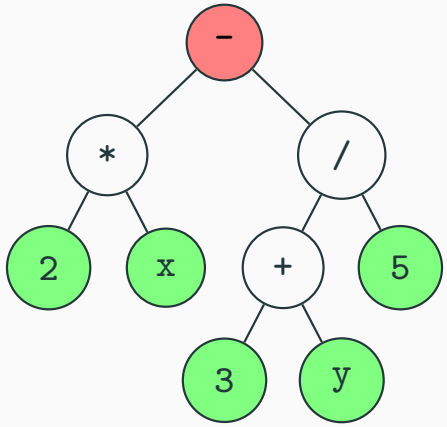


Les arbres binaires d'expression : définitions

- On définit un arbre binaire d'expression (ABE) par récurrence :
- Cas de base : une feuille contenant un nombre ou une lettre est un ABE.
  - L'expression correspondante est le contenu de la feuille.
- Cas composé : L'arbre suivant est une ABE si et seulement si :
  - A_1 et A_2 sont des ABE
  - o une operation (`+`, `-`, `*`, `/`)
  - L'expression associée s'obtient en appliquant o aux expressions associées à A_1 et A_2.



- Exemple : l'arbre suivant correspond au calcul `(2*x)-((3+y)/5)`



Type abstrait : arbres binaires d'expression

- Un arbre est soit un tuple `(r,Ag,Ad)` soit une feuille (int ou str)

- Un arbre est soit un tuple `(r,Ag,Ad)` soit une feuille (int ou str)

abe.py

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

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

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

def fg(A): # A doit être un nœud
    if est_feuille(A):
        raise ValueError("une feuille n'a pas de fils")
    return A[1]

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

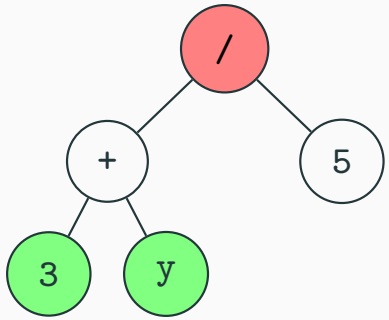
Construction d'un arbre

- Pour construire un arbre, on peut :

Construction d'un arbre

- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

```
A = ('/', ('+', 3, 'y'), 5)
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

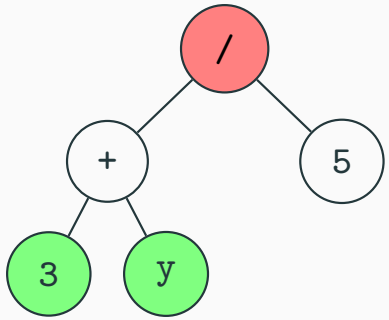
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

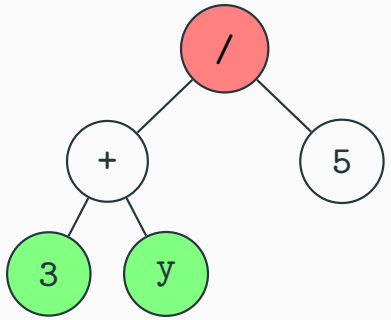
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>>
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

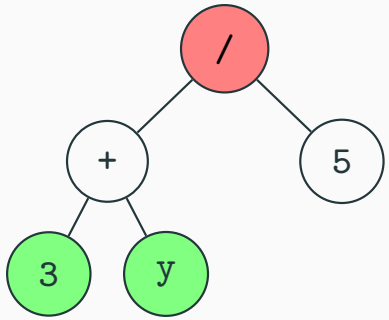
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

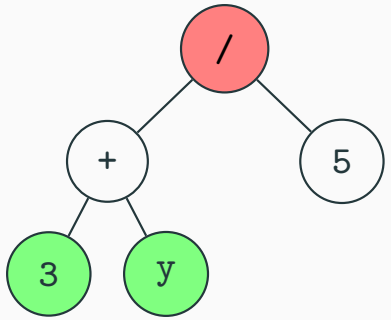
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>>
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

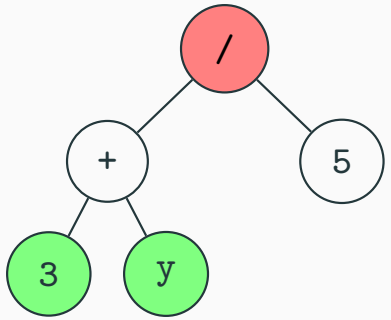
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>> A
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

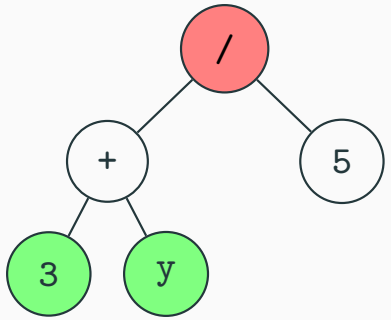
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>> A
('/', ('+', 3, 'y'), 5)
>>>
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

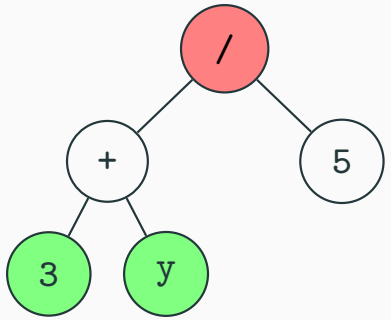
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>> A
('/', ('+', 3, 'y'), 5)
>>> fg(A)
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

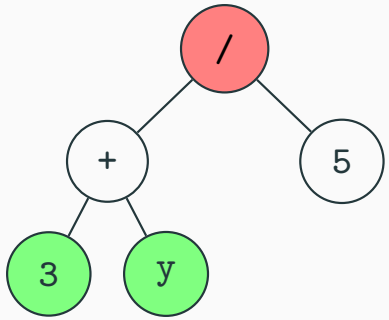
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>> A
('/', ('+', 3, 'y'), 5)
>>> fg(A)
('+', 3, 'y')
>>>
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

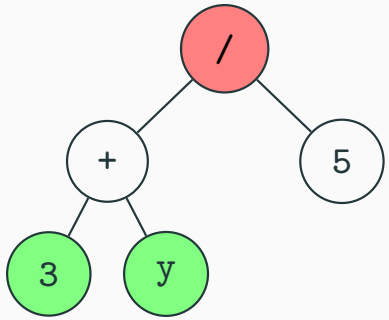
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>> A
('/', ('+', 3, 'y'), 5)
>>> fg(A)
('+', 3, 'y')
>>> fd(fg(A))
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

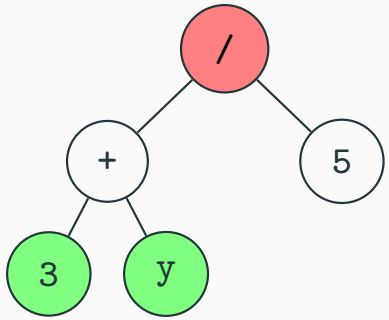
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>> A
('/', ('+', 3, 'y'), 5)
>>> fg(A)
('+', 3, 'y')
>>> fd(fg(A))
'y'
>>>
```



Construction d'un arbre

- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

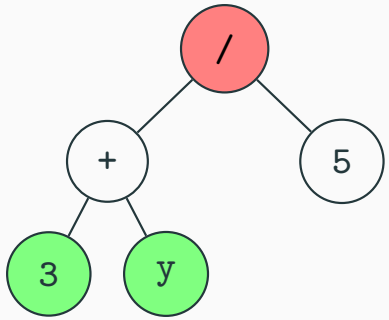
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>> A
('/', ('+', 3, 'y'), 5)
>>> fg(A)
('+', 3, 'y')
>>> fd(fg(A))
'y'
>>> racine(fg(A))
```



- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

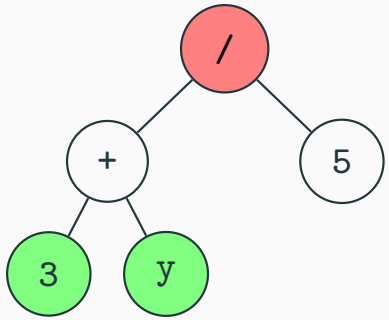
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre('/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>> A
('/', ('+', 3, 'y'), 5)
>>> fg(A)
('+', 3, 'y')
>>> fd(fg(A))
'y'
>>> racine(fg(A))
'+'
```

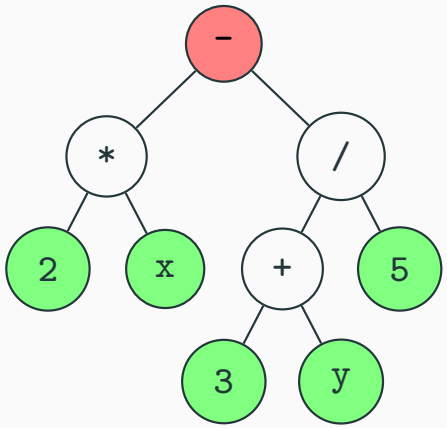


Sommaire

- C'est la plus grande longueur reliant la racine à une feuille

SCRIPT

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



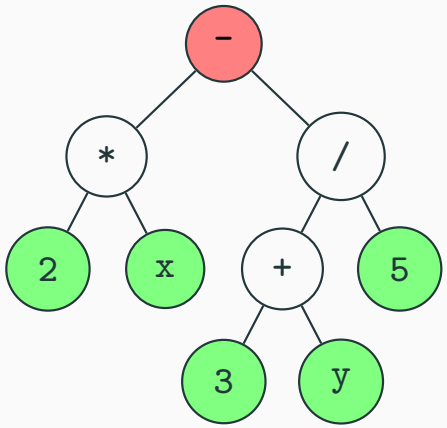
SHELL

```
>>>
```

- C'est la plus grande longueur reliant la racine à une feuille

SCRIPT

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



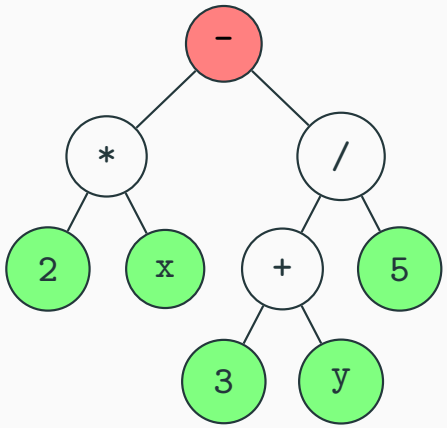
SHELL

```
>>> A
```

- C'est la plus grande longueur reliant la racine à une feuille

SCRIPT

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



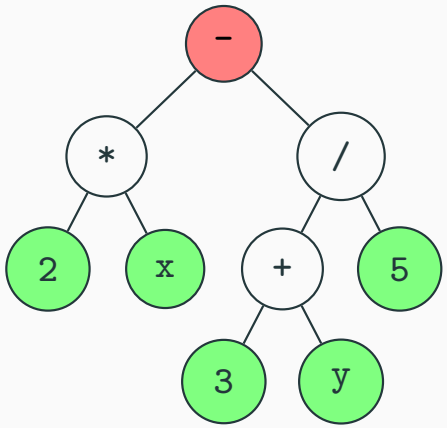
SHELL

```
>>> A
('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))
>>>
```

- C'est la plus grande longueur reliant la racine à une feuille

SCRIPT

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



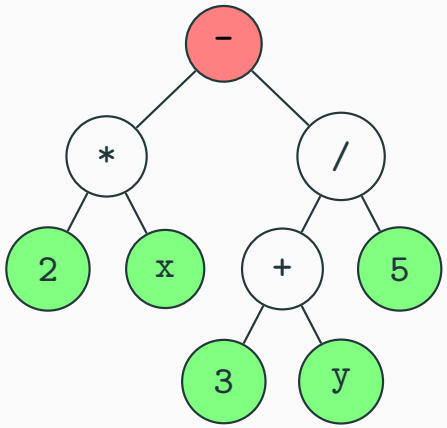
SHELL

```
>>> A
('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))
>>> ( hauteur(A) , hauteur(fg(A)) , hauteur(fd(A)))
```

- C'est la plus grande longueur reliant la racine à une feuille

SCRIPT

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



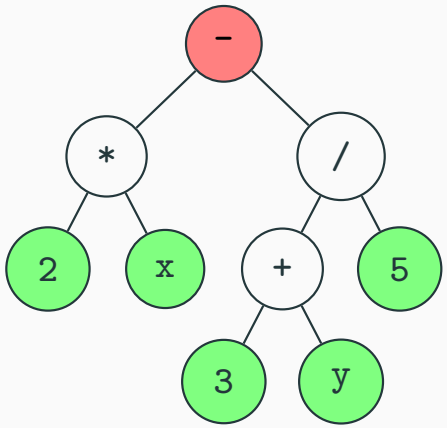
SHELL

```
>>> A
('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))
>>> ( hauteur(A) , hauteur(fg(A)) , hauteur(fd(A)))
(3, 1, 2)
```

- C'est la plus grande longueur reliant la racine à une feuille

SCRIPT

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



SHELL

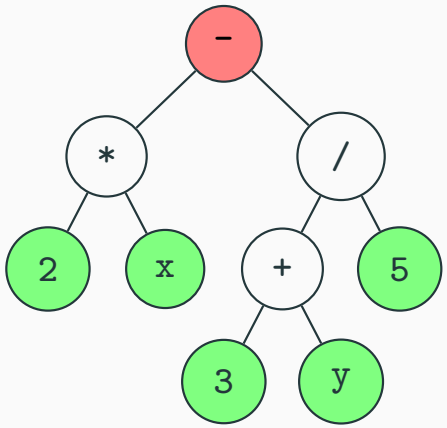
```
>>> A
('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))
>>> ( hauteur(A) , hauteur(fg(A)) , hauteur(fd(A)))
(3, 1, 2)
```

- Notez la récurrence double (`hauteur` est appelée deux fois)

- C'est la plus grande longueur reliant la racine à une feuille

SCRIPT

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



SHELL

```
>>> A
('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))
>>> ( hauteur(A) , hauteur(fg(A)) , hauteur(fd(A)))
(3, 1, 2)
```

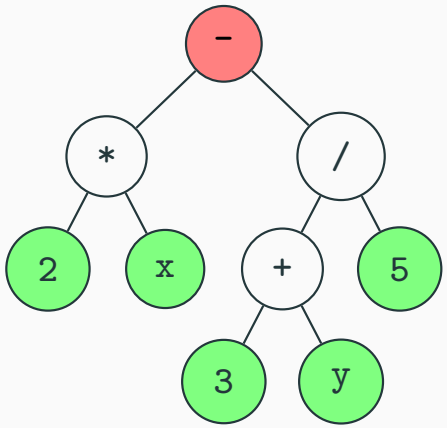
- Notez la récurrence double (`hauteur` est appelée deux fois)
- Un arbre de hauteur h contient au plus 2^h feuilles (*exercice : pourquoi ?*).

Algorithmes simples : Hauteur d'un arbre

- C'est la plus grande longueur reliant la racine à une feuille

SCRIPT

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



SHELL

```
>>> A
('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))
>>> ( hauteur(A) , hauteur(fg(A)) , hauteur(fd(A)))
(3, 1, 2)
```

- Notez la récurrence double (`hauteur` est appelée deux fois)
- Un arbre de hauteur h contient au plus 2^h feuilles (*exercice : pourquoi ?*).
- Réciproquement, pour un arbre de n feuilles, on s'attend à ce qu'il ait une hauteur h telle que $n \approx 2^h$, d'où $h \approx \log_2(n)$.
  - Dans le cas où $n = 2^h$ on parle d'arbre binaire complet.

Algorithmes simples : Présence d'une feuille

- On cherche à savoir si une feuille est présente dans un arbre.

SCRIPT

```
def présente(f, A):
    if est_feuille(A):
        return f == A
    else:
        return présente(f, fg(A)) or présente(f, fd(A))
```

Algorithmes simples : Présence d'une feuille

- On cherche à savoir si une feuille est présente dans un arbre.

SCRIPT

```
def présente(f, A):
    if est_feuille(A):
        return f == A
    else:
        return présente(f, fg(A)) or présente(f, fd(A))
```

- Rappel : le `or` est paresseux.

Algorithmes simples : Présence d'une feuille

- On cherche à savoir si une feuille est présente dans un arbre.

SCRIPT

```
def présente(f, A):
    if est_feuille(A):
        return f == A
    else:
        return présente(f, fg(A)) or présente(f, fd(A))
```

- Rappel : le `or` est paresseux.
  - On cherche d'abord dans le sous-arbre de gauche

- On cherche à savoir si une feuille est présente dans un arbre.

SCRIPT

```
def présente(f, A):
    if est_feuille(A):
        return f == A
    else:
        return présente(f, fg(A)) or présente(f, fd(A))
```

- Rappel : le `or` est paresseux.
  - On cherche d'abord dans le sous-arbre de gauche
  - Et seulement si on n'a pas trouvé, on cherche alors à droite

- On cherche à savoir si une feuille est présente dans un arbre.

SCRIPT

```
def présente(f, A):
    if est_feuille(A):
        return f == A
    else:
        return présente(f, fg(A)) or présente(f, fd(A))
```

- Rappel : le `or` est paresseux.
  - On cherche d'abord dans le sous-arbre de gauche
  - Et seulement si on n'a pas trouvé, on cherche alors à droite

- Le parcours en profondeur consiste à :
  - explorer entièrement le sous-arbre de gauche avant celui de droite.

- On cherche à savoir si une feuille est présente dans un arbre.

SCRIPT

```
def présente(f, A):
    if est_feuille(A):
        return f == A
    else:
        return présente(f, fg(A)) or présente(f, fd(A))
```

- Rappel : le `or` est paresseux.
  - On cherche d'abord dans le sous-arbre de gauche
  - Et seulement si on n'a pas trouvé, on cherche alors à droite

- Le parcours en profondeur consiste à :
  - explorer entièrement le sous-arbre de gauche avant celui de droite.
  - répéter cet ordre d'exploration dans l'exploration des sous-arbres.

Valeur d'un arbre arithmétique

- Un arbre binaire d'expression est dit :

Valeur d'un arbre arithmétique

- Un arbre binaire d'expression est dit :
  - arithmétique si toutes ses feuilles sont des constantes ;

Valeur d'un arbre arithmétique

- Un arbre binaire d'expression est dit :
  - arithmétique si toutes ses feuilles sont des constantes ;
  - algébrique si au moins une feuille est une variable.

- Un arbre binaire d'expression est dit :
  - arithmétique si toutes ses feuilles sont des constantes ;
  - algébrique si au moins une feuille est une variable.

SCRIPT

```
def valeur(A): # l'arbre A doit être arithmétique
    if est_feuille(A):
        return A
    else:
        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
```

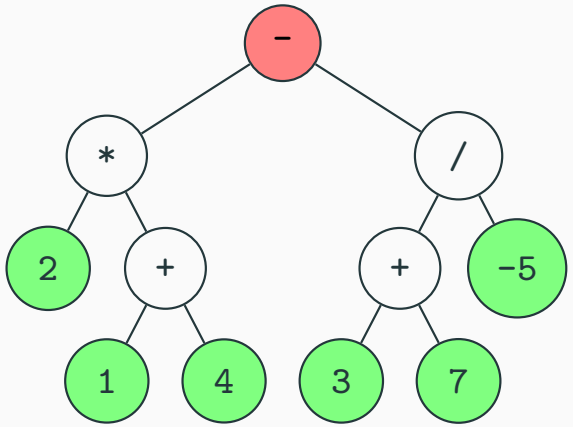
- Un arbre binaire d'expression est dit :
  - arithmétique si toutes ses feuilles sont des constantes ;
  - algébrique si au moins une feuille est une variable.

SCRIPT

```
def valeur(A): # l'arbre A doit être arithmétique
    if est_feuille(A):
        return A
    else:
        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
```

SHELL

```
>>>
```



Valeur d'un arbre arithmétique

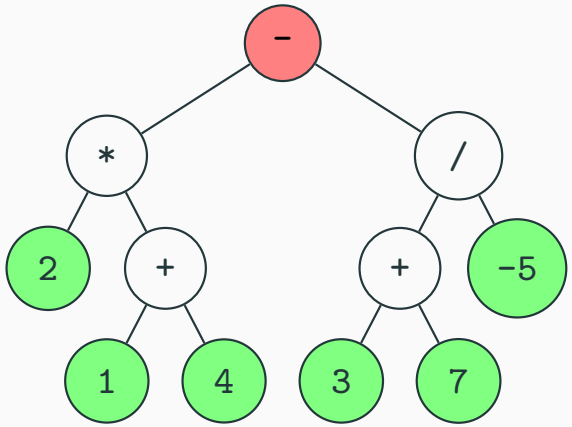
- Un arbre binaire d'expression est dit :
  - arithmétique si toutes ses feuilles sont des constantes ;
  - algébrique si au moins une feuille est une variable.

SCRIPT

```
def valeur(A): # l'arbre A doit être arithmétique
    if est_feuille(A):
        return A
    else:
        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
```

SHELL

```
>>> valeur(fg(A))
```



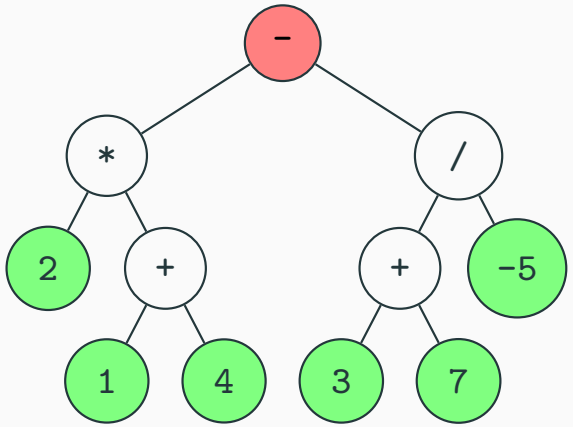
- Un arbre binaire d'expression est dit :
  - arithmétique si toutes ses feuilles sont des constantes ;
  - algébrique si au moins une feuille est une variable.

SCRIPT

```
def valeur(A): # l'arbre A doit être arithmétique
    if est_feuille(A):
        return A
    else:
        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
```

SHELL

```
>>> valeur(fg(A))
10
>>>
```



Valeur d'un arbre arithmétique

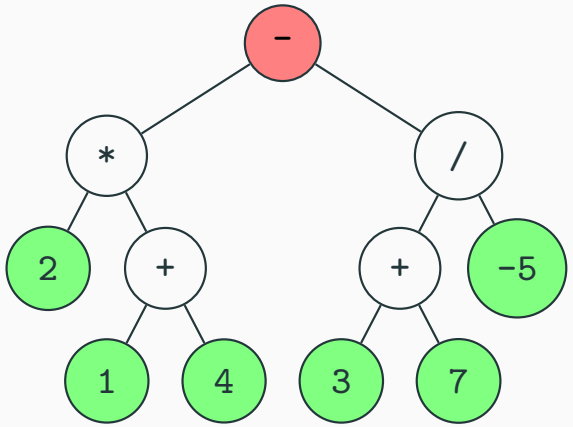
- Un arbre binaire d'expression est dit :
  - arithmétique si toutes ses feuilles sont des constantes ;
  - algébrique si au moins une feuille est une variable.

SCRIPT

```
def valeur(A): # l'arbre A doit être arithmétique
    if est_feuille(A):
        return A
    else:
        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
```

SHELL

```
>>> valeur(fg(A))
10
>>> valeur(A)
```



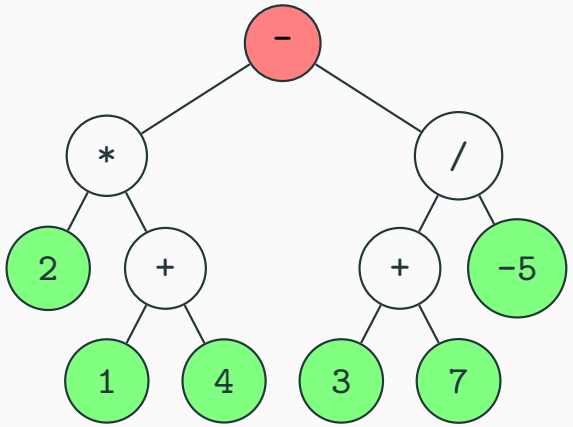
- Un arbre binaire d'expression est dit :
  - arithmétique si toutes ses feuilles sont des constantes ;
  - algébrique si au moins une feuille est une variable.

SCRIPT

```
def valeur(A): # l'arbre A doit être arithmétique
    if est_feuille(A):
        return A
    else:
        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
```

SHELL

```
>>> valeur(fg(A))
10
>>> valeur(A)
12.0
```



Pile et parcours itératif : principe

- Les arbres formant un type récursif,
  - il est naturel de programmer récursivement pour les manipuler.

Pile et parcours itératif : principe

- Les arbres formant un type récursif,
  - il est naturel de programmer récursivement pour les manipuler.
- Mais on peut aussi produire un algorithme itératif (avec une boucle).

Pile et parcours itératif : principe

- Les arbres formant un type récursif,
  - il est naturel de programmer récursivement pour les manipuler.
- Mais on peut aussi produire un algorithme itératif (avec une boucle).
- On utilise une pile pour stocker les branches qu'il nous reste à visiter.
  - Quand on tombe sur un sous-arbre à visiter, à l'ajoute à la pile.
  - On traite les sous-arbres du haut de la pile en premier

Pile et parcours itératif : principe

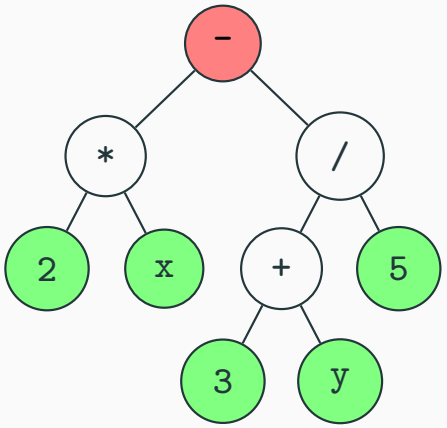
- Les arbres formant un type récursif,
  - il est naturel de programmer récursivement pour les manipuler.
- Mais on peut aussi produire un algorithme itératif (avec une boucle).
- On utilise une pile pour stocker les branches qu'il nous reste à visiter.
  - Quand on tombe sur un sous-arbre à visiter, à l'ajoute à la pile.
  - On traite les sous-arbres du haut de la pile en premier
- Exemple : calcul du nombre de feuilles de A en itératif.
  - Initialisation : je mets l'arbre dans une pile vide
  - À chaque tour de boucle je mets le sommet de la pile dans la variable `A`
  - si `A` est un arbre composé : il me reste à visiter les deux fils
    - j'empile le fils droit de `A`
    - j'empile le fils gauche de `A`
  - si `A` est un arbre simple (une feuille)
    - J'ajoute `1` au compteur de feuille

SCRIPT

```
def nb_feuilles(A): # le nombre de feuilles
    P = nouvelle_pile() ; empile(P, A)
    res = 0
    # tant qu'il reste des sous-arbres à visiter
    while not est_vide(P):
        print(P)
        A = dépile(P)
        if est_feuille(A):
            res = res + 1
        else:
            empile(P, fd(A)); empile(P, fg(A))
    return res
```

SHELL

```
>>>
```



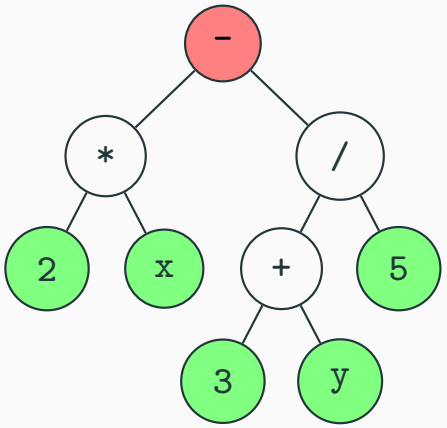
Pile et parcours itératif : code

SCRIPT

```
def nb_feuilles(A): # le nombre de feuilles
    P = nouvelle_pile() ; empile(P, A)
    res = 0
    # tant qu'il reste des sous-arbres à visiter
    while not est_vide(P):
        print(P)
        A = dépile(P)
        if est_feuille(A):
            res = res + 1
        else:
            empile(P, fd(A)); empile(P, fg(A))
    return res
```

SHELL

```
>>> nb_feuilles(A)
```



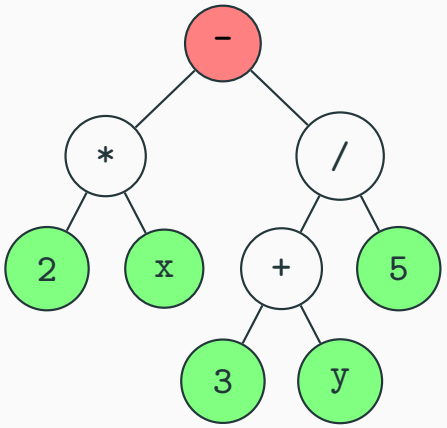
Pile et parcours itératif : code

SCRIPT

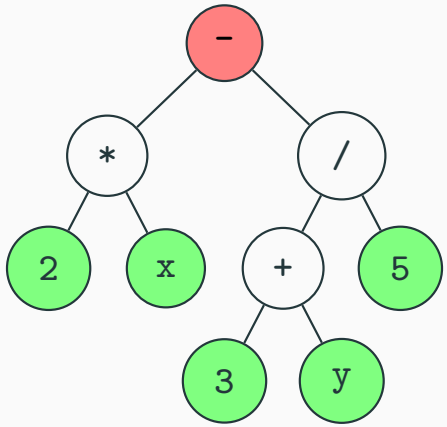
```
def nb_feuilles(A): # le nombre de feuilles
    P = nouvelle_pile() ; empile(P, A)
    res = 0
    # tant qu'il reste des sous-arbres à visiter
    while not est_vide(P):
        print(P)
        A = dépile(P)
        if est_feuille(A):
            res = res + 1
        else:
            empile(P, fd(A)); empile(P, fg(A))
    return res
```

SHELL

```
>>> nb_feuilles(A)
[('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))]
[('/', ('+', 3, 'y'), 5), ('*', 2, 'x')]
[('/', ('+', 3, 'y'), 5), 'x', 2]
[('/', ('+', 3, 'y'), 5), 'x']
[('/', ('+', 3, 'y'), 5)]
[5, ('+', 3, 'y')]
[5, 'y', 3]
[5, 'y']
[5]
5
```



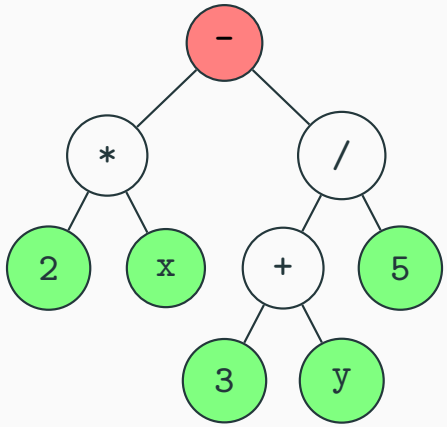
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=` `res=0`

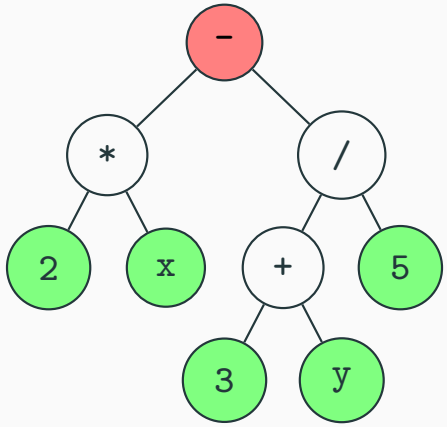
```
('-',('*',2,'x'),('/',('+',3,'y'),5))
```

- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre




```
A=('-',('*',2,'x') ,('/',('+',3,'y'),5))                    res=0
```

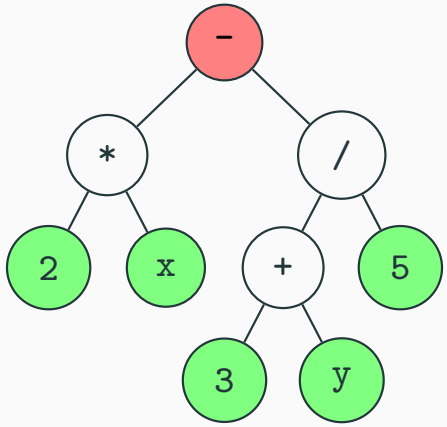
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=` `res=0`

`('*',2,'x')`
`('/',('+',3,'y'),5)`

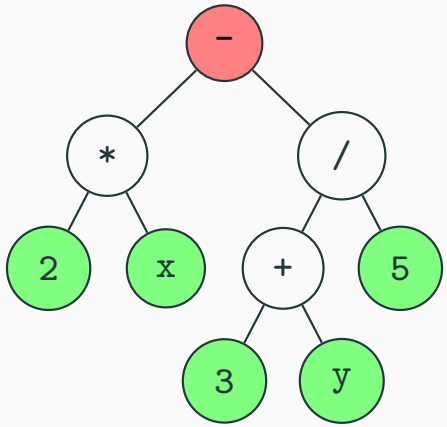
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=` `res=0`

`('*',2,'x')`
`('/',('+',3,'y'),5)`

- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre

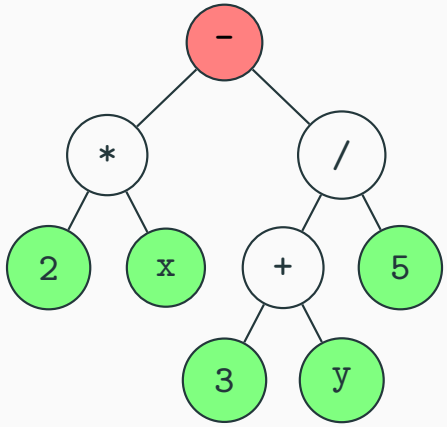



```
A=('*',2 ,'x')
```

```
res=0
```

```
('/',('+',3,'y'),5)
```

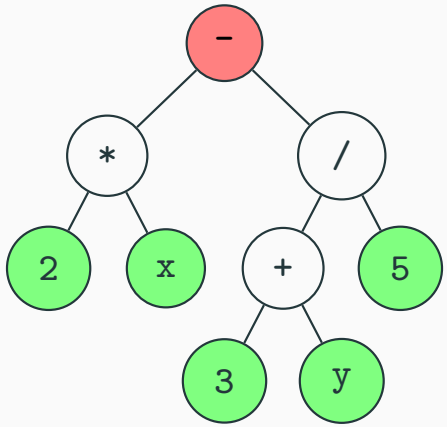
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=` `res=0`

2
`'x'`
`('/',('+',3,'y'),5)`

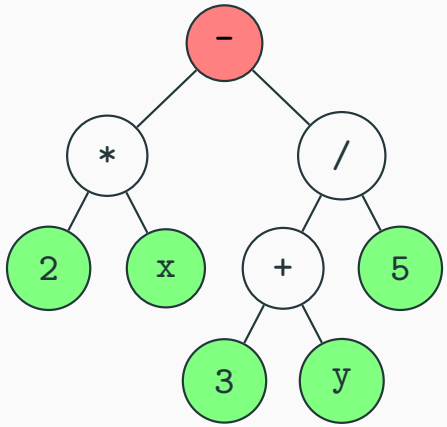
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=` `res=0`

2
'x'
('/',('+',3,'y'),5)

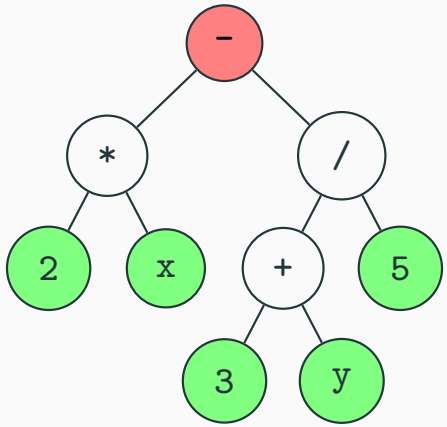
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=2` `res=1`

`'x'`
`('/',('+',3,'y'),5)`

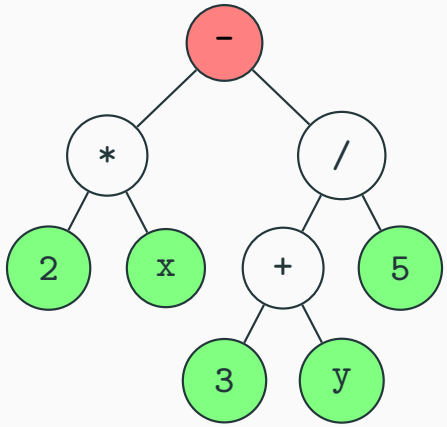
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A='x'` `res=2`

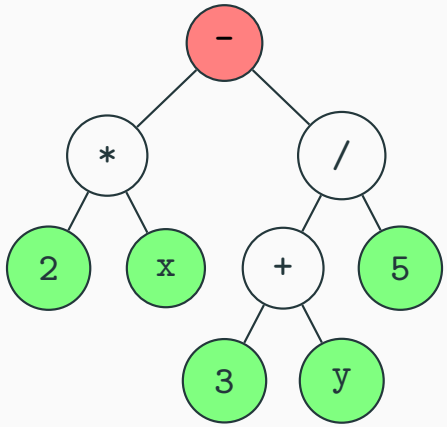
```
('/',('+',3,'y'),5)
```

- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre




```
A=('/',('+',3,'y') ,5)                                    res=2
```

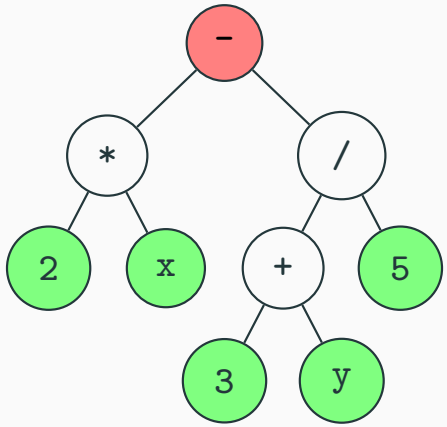
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=` `res=2`

`('+',3,'y')`
`5`

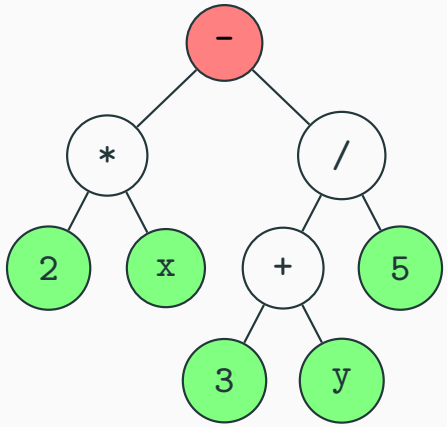
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=` `res=2`

`('+',3,'y')`
`5`

- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre

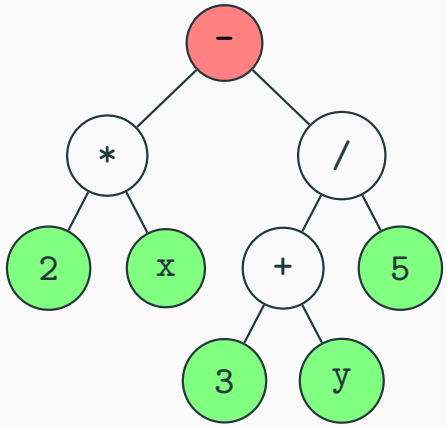



```
A=('+',3 ,'y')
```

```
res=2
```

5

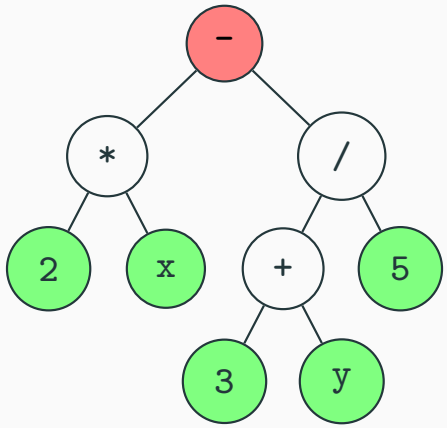
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=` `res=2`

3
'y'
5

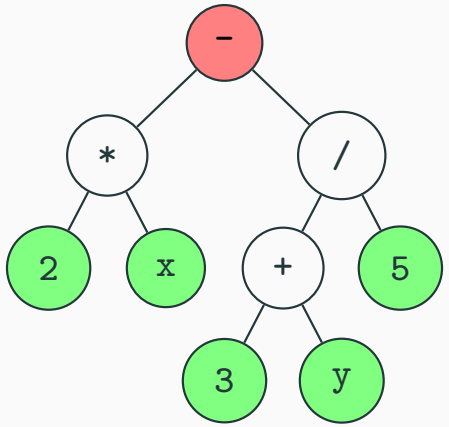
- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=` `res=2`

3
'y'
5

- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre

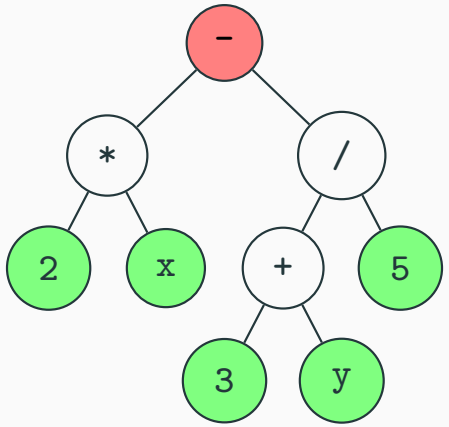


`A=3`

`res=3`

`'y'`
`5`

- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre

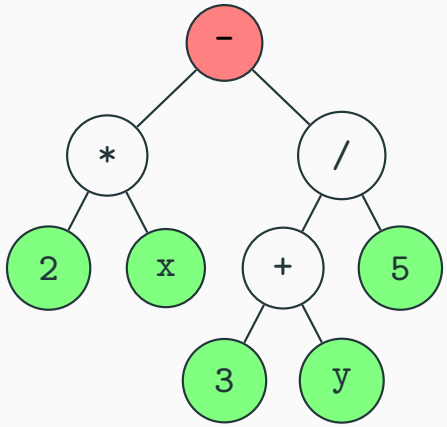


`A='y'`

`res=4`

5

- Pour bien comprendre détaillons l'exemple précèdent.
- Prenez le temps de bien le comprendre



`A=5` `res=5`

Sommaire

- Pour l'instant la pile est implémentée avec un tableau
- Les tableaux se prêtent bien aux algorithmes itératifs

- Pour l'instant la pile est implémentée avec un tableau
- Les tableaux se prêtent bien aux algorithmes itératifs
- Peux-on concevoir une pile de manière récursive ?

- Pour l'instant la pile est implémentée avec un tableau
- Les tableaux se prêtent bien aux algorithmes itératifs
- Peux-on concevoir une pile de manière récursive ?
- Il existe un "équivalent" récursif : la liste chaînée.
- Attention :
  - Tableau : taille fixe et programmation impérative (en Python : liste)
  - Liste chaînée : taille dynamique et programmation récursive

Définition inductive

- On va donner un définition récursive des listes chaînée :

- On va donner un définition récursive des listes chaînée :
- Cas de base : la valeur `None` est une liste chaînée
  - c'est évidemment la liste vide

- On va donner un définition récursive des listes chaînée :
- Cas de base : la valeur `None` est une liste chaînée
  - c'est évidemment la liste vide
- Cas récursif : une couple `(tête,queue)` avec :
  - `tête` : premier élément de la liste.
  - `queue` : un autre liste chaînée (la suite de la liste)

- On va donner un définition récursive des listes chaînée :
- Cas de base : la valeur `None` est une liste chaînée
  - c'est évidemment la liste vide
- Cas récursif : une couple `(tête,queue)` avec :
  - `tête` : premier élément de la liste.
  - `queue` : un autre liste chaînée (la suite de la liste)
- Exemple de liste chaînée contenant les valeurs 1, 2 et 3

SCRIPT
```
LC = (1, (2, (3, None)))
```

- Contrairement au pile, on ne modifie pas les données.

- Contrairement au pile, on ne modifie pas les données.

lc.py

```
def is_empty(lc):
    return lc == None

def head(lc):
    (h,t) = lc
    return h

def tail(lc):
    (h,t) = lc
    return t
```

- On s'obligera comme pour les arbres à n'utiliser que l'interface abstraite.
- On s'autorisera par contre la notation de couple `(a,b)` pour construire un nouvelle chaîne.

- On souhaite calculer la somme

- On souhaite calculer la somme

SCRIPT

```
def somme(lc):
    if is_empty(lc):
        return 0
    else:
        return head(lc) + somme(tail(lc))
```

SHELL

```
>>>
```

Exemples : la sommes des éléments

- On souhaite calculer la somme

SCRIPT

```
def somme(lc):
    if is_empty(lc):
        return 0
    else:
        return head(lc) + somme(tail(lc))
```

SHELL

```
>>> L = (100, (20, (3, None)))
```

- On souhaite calculer la somme

SCRIPT

```
def somme(lc):
    if is_empty(lc):
        return 0
    else:
        return head(lc) + somme(tail(lc))
```

SHELL

```
>>> L = (100, (20, (3, None)))
>>>
```

- On souhaite calculer la somme

SCRIPT

```
def somme(lc):
    if is_empty(lc):
        return 0
    else:
        return head(lc) + somme(tail(lc))
```

SHELL

```
>>> L = (100, (20, (3, None)))
>>> somme(L)
```

- On souhaite calculer la somme

SCRIPT

```
def somme(lc):
    if is_empty(lc):
        return 0
    else:
        return head(lc) + somme(tail(lc))
```

SHELL

```
>>> L = (100, (20, (3, None)))
>>> somme(L)
123
```

Exemples : Affichage

- On souhaite transformer la liste en chaîne `"1 → 2 → 3 → ."`

Exemples : Affichage

- On souhaite transformer la liste en chaîne `"1 → 2 → 3 → ."`

SCRIPT

```
def vers_string(lc):
    if is_empty(lc):
        return "."
    else:
        return str(head(lc)) + " → " + vers_string(tail(lc))
```

SHELL

```
>>>
```

Exemples : Affichage

- On souhaite transformer la liste en chaîne `"1 → 2 → 3 → ."`

SCRIPT

```
def vers_string(lc):
    if is_empty(lc):
        return "."
    else:
        return str(head(lc)) + " → " + vers_string(tail(lc))
```

SHELL

```
>>> L = (100, (20, (3, None)))
```

Exemples : Affichage

- On souhaite transformer la liste en chaîne "1 → 2 → 3 → ."

SCRIPT

```
def vers_string(lc):
    if is_empty(lc):
        return "."
    else:
        return str(head(lc)) + " → " + vers_string(tail(lc))
```

SHELL

```
>>> L = (100, (20, (3, None)))
>>>
```

Exemples : Affichage

- On souhaite transformer la liste en chaîne `"1 → 2 → 3 → ."`

SCRIPT

```
def vers_string(lc):
    if is_empty(lc):
        return "."
    else:
        return str(head(lc)) + " → " + vers_string(tail(lc))
```

SHELL

```
>>> L = (100, (20, (3, None)))
>>> print(vers_string(L))
```

- On souhaite transformer la liste en chaîne `"1 → 2 → 3 → ."`

SCRIPT

```
def vers_string(lc):
    if is_empty(lc):
        return "."
    else:
        return str(head(lc)) + " → " + vers_string(tail(lc))
```

SHELL

```
>>> L = (100, (20, (3, None)))
>>> print(vers_string(L))
100 → 20 → 3 → .
```

Merci pour votre attention
Questions

?

Cours bonus — Types récursifs en Python

Licence