Séance 7 : Modules et types abstraits

L1 – Université Côte d'Azur

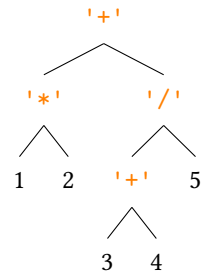
Dans les exercices suivants, on manipule des arbres binaires d'expression arithmétiques (non vides), comme définis dans le cours. Les expressions seront des nombres entiers (sur les feuilles de l'arbre) ou des chaînes de caractères contenant les opérations (sur les nœuds de l'arbre).

On ne s'intéresse pas à la façon dont les arbres sont représentés, on va utiliser uniquement les fonctions suivantes :

- `arbre(r, Ag, Ad)` qui renvoie un arbre de racine `r` et de fils `Ag` (gauche) et `Ad` (droit) ;
- `est_feuille(obj)` qui renvoie `True` si `obj` est une feuille, et `False` sinon ;
- `racine(A)` qui renvoie la racine de l'arbre `A`
- `fg(A)` qui renvoie le fils gauche de l'arbre `A`
- `fd(A)` qui renvoie le fils droit de l'arbre `A`.

Exercice 1 — Mise en route (⋆)

1. Téléchargez le fichier `abe.py` à l'adresse https://upinfo.univ-cotedazur.fr/~obaldellon/L1/py/tp7/abe.py Placez-le dans le répertoire où vous écrivez vos TP. Créez un fichier `tp7-arbres.py`.
   Quelle ligne faut-il écrire dans ce fichier pour pouvoir utiliser les fonctions définies dans `abe.py` ?
2. Sans utiliser de liste, mais uniquement la fonction `arbre(r, Ag, Ad)`, créez l'arbre `A1` défini ci-contre.



Exercice 2 — Manipulation d'arbres (⋆⋆)

1. Écrire une fonction `contient42(A)` qui renvoie `True` si une des feuilles de l'arbre `A` est 42, et `False` sinon.
   Une première version :

```python
def contient42(A):
    if est_feuille(A):
        return A == 42
    else:
        résultat_gauche = contient42(fg(A))
        résultat_droit = contient42(fd(A))
        return résultat_gauche or résultat_droit
```

En plus efficace, car le or est paresseux

```
def contient42(A):
    if est_feuille(A):
        return A == 42
    else:
        return contient42(fg(A)) or contient42(fd(A))
```

2. Écrire une fonction `compte_pairs(A)` qui renvoie le nombre de feuilles de `A` qui sont des nombres pairs.

```
def compte_pairs(A):
    if est_feuille(A):
        if A%2 == 0:
            return 1
        else:
            return 0
    else:
        return compte_pairs(fg(A)) + compte_pairs(fd(A))
```

3. Écrire une fonction `feuilles_paires(A)` qui renvoie la liste des feuilles de `A` qui sont des nombres pairs.

```
def feuilles_paires(A):
    if est_feuille(A):
        if A%2 == 0:
            return [A]
        else:
            return []
    else:
        g = feuilles_paires(fg(A))
        d = feuilles_paires(fd(A))
        return g + d
```

On appelle *profondeur* d'un nœud la distance entre ce nœud et la racine.

4. Écrire une fonction `liste_profondeur(A,n)` qui renvoie la liste des nœuds de `A` de profondeur `n`

```
def liste_profondeur(A, n):
    if est_feuille(A):
        return []
    else:
        if n == 0:
            return [racine(A)]
        else:
            g = liste_profondeur(fg(A), n-1)
            d = liste_profondeur(fd(A), n-1)
            return g + d
```

5. Écrire une fonction `profondeur_max_pairs(A)` qui renvoie la profondeur maximale des feuilles paires de `A` (et `None` si `A` n'a pas de feuille paire).

```python
def profondeur_max_pair(A):
    if est_feuille(A):
        if A%2==0:
            return 0
        else:
            return None
    else:
        rg = profondeur_max_pair(fg(A))
        rd = profondeur_max_pair(fd(A))
        if rg==None and rd==None:
            return None
        elif rg==None: # rd est bien définie
            return rd+1
        elif rd==None: # rg est bien définie
            return rg+1
        else: # rg et rd sont bien définie
            return max(rg,rd)+1 # on ajoute 1 à la profondeur maximale trouvée
```

Exercice 3 — Comptage et transformation (⋆⋆)

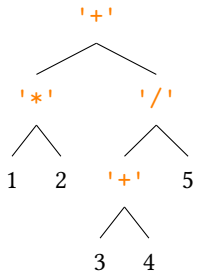
1. Programmez une fonction `compter(A, op)` renvoyant le nombre d'apparitions de l'opérateur op dans l'arbre `A`.

```
>>> compter(A1, '+')
2
```

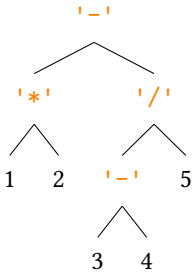
```python
def compter(A, op):
    if est_feuille(A):
        return 0
    else:
        rg = compter(fg(A),op)
        rd = compter(fd(A),op)
        if racine(A) == op:
            return 1 + rg + rd
        else:
            return  rg + rd
```

2. Programmez une fonction `remplacer(A, op1, op2)` qui renvoie une copie de l'arbre `A` où chaque apparition de l'opérateur `op1` aura été remplacée par l'opérateur `op2`.

Arbre A1



Arbre A2 en posant `A2=remplacer(A1,'+','-')`

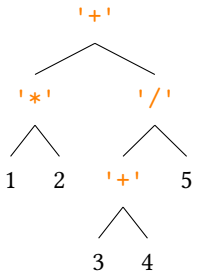



```
def remplacer(A, op1, op2):
    if est_feuille(A):
        return A
    else:
        Ag = remplacer(fg(A),op1,op2)
        Ad = remplacer(fd(A),op1,op2)
        if racine(A)==op1:
            op=op2
        else:
            op=racine(A)
        return arbre(op,Ag,Ad)
```

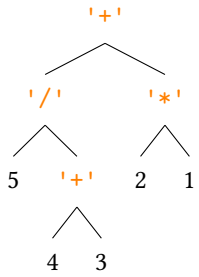
Exercice 4 — Miroir (⋆⋆)

Programmez la fonction `miroir(A)` retournant l'image inversée (à tous les niveaux) de l'arbre `A`.

Arbre A1



Miroir de A1




```
def miroir(A):
    if est_feuille(A):
        return A
    else:
        return arbre(racine(A), miroir(fd(A)), miroir(fg(A)))
```

Exercice 5 — Parcours suffixe et évaluation avec une pile (⋆ ⋆ ⋆)

Le but de cette exercice est d'évaluer des expressions arithmétiques automatiquements. Certes, Python sait déjà le faire, mais il peut intéressant de savoir le programmer soi-même.

Il est assez difficile de transformer une expression représentée par une chaîne (par exemple `"1+(4+5*3)"`) en arbre. Il faut gérer les priorités et les parenthèses. Il existe cependant une écriture, la fameuse *notation polonaise inversée* (NPI), très simple à transformer en arbre. C'était la notation utilisée par les calculatrices HP [1] lors de la jeunesse de mon père (ce

1. https://fr.wikipedia.org/wiki/Calculatrices_HP

qui ne nous rajeunit pas...). La NPI s'obtient par un parcours en profondeur suffixe. Par exemple, le parcours profondeur suffixe de l'expression `(1-2)*(3+4)` est

```
[1, 2, '-', 3, 4, '+', '*']
```

1. Écrivez un programme `arboriser` qui prend une liste en NPI et renvoie l'arbre correspondant. On utilisera un pile pour stocker les résultats intermédiaires. Pour information vous pouvez télécharger le module `pile.py` (celui du cours) à l'adresse : https://upinfo.univ-cotedazur.fr/~obaldellon/L1/py/tp7/pile.py

```python
import pile

def arboriser(L):
    P = pile.nouvelle_pile()
    for e in L :
        if e in "+-*/":
            a = pile.dépile(P)
            b = pile.dépile(P)
            pile.empile(P,arbre(e,b,a))
        else:
            pile.empile(P,e)
    return pile.dépile(P) # Si le calcul a une bonne syntaxe, il ne
                          # doit rester qu'un élément dans la pile
```

2. En utilisant la fonction `arboriser` et la fonction `valeur` (cours 7 page 43) qui calcule la valeur associée à un arbre binaire arithmétique, écrire une fonction `calcul` qui donne le résultat du calcul donnée en argument en notation NPI.

```
>>> calcul([1, 2, '-', 3, 4, '+', '*'])
-7
```

```python
def valeur(A):
    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


def calcul(L):
    return valeur(arboriser(L))
```

3. Écrire une fonction `calcul_direct` qui fait la même chose que la fonction `calcul` mais sans passer par un arbre et en faisant le calcul directement avec une pile.

```python
def calcul_simple(e,a,b):
    if e == '+' : return a+b
    if e == '-' : return a-b
    if e == '*' : return a*b
    if e == '/' : return a/b


def calcul_direct(L):
    P = pile.nouvelle_pile()
    for e in L :
        if est_opérateur(e):
            a = pile.dépile(P)
            b = pile.dépile(P)
            pile.empile(P,calcul_simple(e,b,a)) # il suffit de changer cette ligne !
        else:
            pile.empile(P,e)
    return pile.dépile(P) # Si le calcul a une bonne syntaxe, il ne
                          # doit rester qu'un élément dans la pile
```

4. Écrire une fonction `calcul_chaîne` qui calcule la valeur d'une chaîne en NPI.

```
>>> calcul_chaîne('1 2 - 3 4 + *')
-7
```

```python
def calcul_chaine(s):
    Ls = s.split()
    L = []
    for e in Ls:
        if e.isdigit():
            L.append(int(e))
        else:
            L.append(e)
    return calcul_direct(L)
```