

# Numérique et sciences informatiques - Devoir 1 - corrigé

## Exercice 1 : Évaluation d'expression à l'aide d'arbres (10 points)

On s'intéresse à l'évaluation d'expressions mathématiques **comportant uniquement des additions et des multiplications**. On utilisera pour cela les structures de file et de pile, dont les interfaces sont données ci-dessous :

### Interface de la classe `Pile` :

- `Pile()` : crée une pile vide.
- `empile(e1)` : empile l'élément `e1` au sommet de la pile.
- `depile()` : supprime et renvoie l'élément au sommet de la pile, déclenche une erreur si la pile est vide.
- `est_vide()` : renvoie `True` si la pile est vide, `False` sinon.

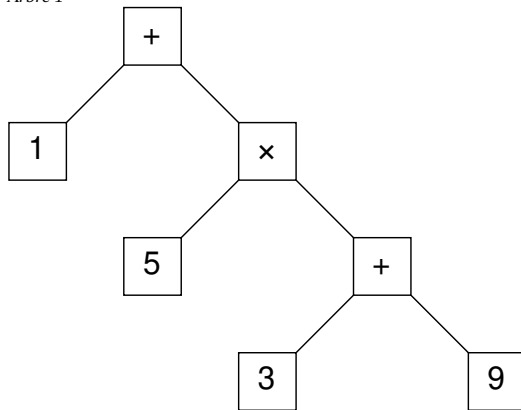
### Interface de la classe `File` :

- `File()` : crée une file vide.
- `enfile(e1)` : enfile l'élément `e1` à la queue de la file.
- `defile()` : supprime et renvoie l'élément en tête de la file, déclenche une erreur si la file est vide.
- `est_vide()` : renvoie `True` si la file est vide, `False` sinon.

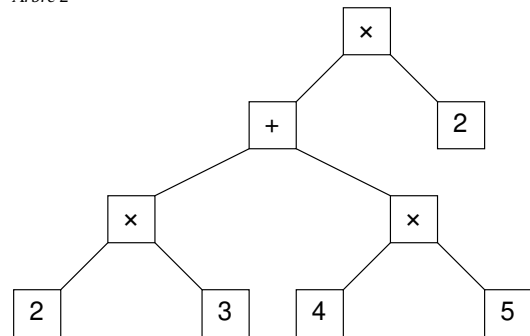
Pour tenir compte de l'ordre des opérations, on représente les expressions par des arbres binaires.

Ainsi, l'expression  $1+5\times(3+9)$  est représentée par l'arbre 1 :


Arbre 1



Arbre 2



1. Donner l'expression représentée par l'arbre 2.

 (1 pt)  $((2\times 3)+(4\times 5))\times 2$

2. On décide d'implémenter en Python un arbre binaire à l'aide de la classe `Noeud` ci-dessous :

```
1 class Noeud:
2     def __init__(self, etiquette, gauche, droit):
3         self.etiq = etiquette
4         self.sag = gauche
5         self.sad = droit
```

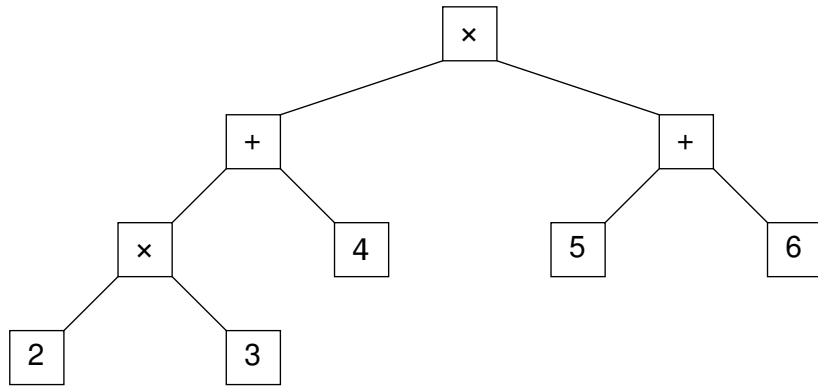
Un sous-arbre vide sera représenté par `None`.

Dessiner l'arbre `expression` qui est défini par le code suivant :

```
1 feuille2 = Noeud("2", None, None)
2 feuille3 = Noeud("3", None, None)
3 feuille4 = Noeud("4", None, None)
4 feuille5 = Noeud("5", None, None)
5 feuille6 = Noeud("6", None, None)
6 noeud0 = Noeud("*", feuille2, feuille3)
7 noeud1 = Noeud("+", feuille5, feuille6)
```

```
noeud2 = Noeud("+", noeud0, feuille4)
expression = Noeud("*", noeud2, noeud1)
```

✎ (1 pt)  $((2 \times 3) + 4) \times (5 + 6)$



3. Le parcours suffixe (aussi appelé postfixe) d'un arbre représentant une expression mathématique permet d'en obtenir une représentation appelée *notation polonaise inversée*.

- a. Donner la liste des étiquettes de l'arbre 2 de la question 1 dans l'ordre du parcours suffixe de cet arbre.

✎ (2 pts) ["2", "3", "x", "4", "5", "x", "+", "2", "x"]

- b. On donne ci-après trois propositions de fonctions récursives dont le premier paramètre est un arbre représentant une expression mathématique et le deuxième est une file initialement vide. Laquelle de ces fonctions renvoie la file contenant les étiquettes de l'arbre dans l'ordre du parcours suffixe ? Justifier.

**Proposition 1**

```
1 def suffixe(arbre, file):
2     if arbre != None
3         file.enqueue(arbre.etiq)
4         parcours_g = suffixe(arbre.sag, file)
5         parcours_d = suffixe(arbre.sad, file)
6     return file
```

**Proposition 2**

```
1 def suffixe(arbre, file):
2     if arbre != None
3         parcours_g = suffixe(arbre.sag, file)
4         file.enqueue(arbre.etiq)
5         parcours_d = suffixe(arbre.sad, file)
6     return file
```

**Proposition 3**

```
1 def suffixe(arbre, file):
2     if arbre != None
3         parcours_g = suffixe(arbre.sag, file)
4         parcours_d = suffixe(arbre.sad, file)
5         file.enqueue(arbre.etiq)
6     return file
```

✎ (2 pts) Il s'agit de la proposition 3 ; suffixe : la racine est notée à la fin.

4. L'évaluation d'une expression mathématique consiste à effectuer les différentes opérations pour obtenir le résultat du calcul correspondant.  
On donne un algorithme qui permet d'évaluer une expression sous la forme d'un arbre :

- On effectue un parcours suffixe de cet arbre pour obtenir une file contenant ses étiquettes dans l'ordre de la notation polonaise inversée.
- Pour chaque élément défilé :
  - Si c'est un nombre : on l'empile.
  - Si c'est un opérateur (" $+$ " ou " $*$ "), on dépile les deux éléments  $d$  et  $g$  au sommet de la pile, et on empile le **résultat** de l'opération appliquée à  $g$  et  $d$ .
- Lorsque la file est vide, la pile contient un seul élément : le résultat de l'évaluation de l'expression.

Par exemple, lors de l'évaluation de l'expression en notation polonaise inversée :

`3 10 + 5 *`

voici les différents états de la pile, suite au défilement d'un élément :

Élément défilé	3	10	+	5	*
Pile	3	10 3	13	5 13	65

La fonction `evalue(arbre)` implémente cet algorithme.

Elle renvoie le résultat de l'évaluation d'une expression mathématique représentée par un arbre binaire `arbre` passé en paramètre.

Compléter cette fonction. On pourra utiliser la fonction `op` définie ci-contre.

```

1  def op(symbole, x, y):
2      if symbole == '+':
3          return int(x) + int(y)
4      else:
5          return int(x) * int(y)

```

 (4 pts)

```

1  def evalue(arbre):
2      f = suffixe(arbre, File())
3      p = Pile()
4      while not(f.est_vide()) :
5          elt = f.defile()
6          if elt == '+' or elt == '*':
7              x = p.depile()
8              y = p.depile()
9              resultat = op(elt, x, y)
10             p.empile(resultat)
11
12         else:
13             p.empile(elt)
14
15     return p.depile()

```

## Exercice 2 : SQL : Jeu de Go (10 points)

Le jeu de go est un jeu de société originaire de Chine. Il oppose deux adversaires qui placent à tour de rôle des pierres, respectivement noires et blanches, sur un plateau.

Dans cette partie on pourra utiliser les mots clés suivants du langage SQL :

`SELECT`, `INSERT INTO`, `DISTINCT`, `WHERE`, `UPDATE`, `JOIN`, `COUNT`, `MIN`, `MAX`, `ORDER BY`.

- La fonction d'agrégation `COUNT(*)` renvoie le nombre d'enregistrements de la requête.
- Les fonctions d'agrégations `MIN(propriete)` et `MAX(propriete)` renvoient respectivement la plus petite et la plus grande valeur de l'attribut `propriete` pour les enregistrements de la requête.
- La commande `ORDER BY propriete` permet de trier les résultats d'une requête selon l'attribut `propriete`.

Le responsable de la fédération internationale de go enregistre dans une base de données les résultats de parties historiques.

Il définit pour cela des relations `Joueurs`, `Parties` et `Tournois` qui suivent le schéma relationnel suivant (les clés primaires sont soulignées et les clés étrangères sont précédées du caractère #).

`idnoir` et `idblanc` permettent de repérer quel joueur avait quelle couleur de pierre (blanches ou noires).

Joueurs	
<u>idjoueur</u>	INT
nom	TXT
naissance	INT
nation	TXT

Parties	
# <u>idnoir</u>	INT
# <u>idblanc</u>	INT
# <u>tournoi</u>	INT
jour	DATE
score	FLOAT

Tournois	
<u>idtournoi</u>	INT
nom	TXT
pays	TXT

1. On suppose que ce schéma relationnel a été implémenté dans le système de gestion de bases de données. Voici trois lignes extraites du script SQL ayant servi à la création de ce schéma :

- FOREIGN KEY (idnoir) REFERENCES Joueurs(idjoueur)
- FOREIGN KEY (idblanc) REFERENCES Joueurs(idjoueur)
- FOREIGN KEY (tournoi) REFERENCES Tournoi(idtournoi)

Tracer, sur le schéma relationnel, des flèches reliant chaque clé étrangère à l'attribut qu'elle référence.

2. La base de données est vide et on souhaite enregistrer les résultats d'un premier tournoi à l'aide des commandes SQL suivantes :

```

1  INSERT INTO Joueurs (idjoueur, nom, naissance, nation)
2  VALUES (1, 'Dosaku', 1645, 'Japon'),
3          (2, 'Genan Inseki', 1798, 'Japon'),
4          (3, 'Shusaku', 1829, 'Japon');
5
6  INSERT INTO Parties (idnoir, idblanc, tournoi, jour, score)
7  VALUES (2, 3, 1, '1846-09-12', -2);
8
9  INSERT INTO Tournoi (idtournoi, nom, pays)
10 VALUES (1, 'Osaka', 'Japon')

```

a. Quelle est l'erreur produite par cette suite d'instructions ? Justifier.

(2 pts)

Il est impossible d'insérer la partie correspondant au tournoi 1 dans la relation

Parties sachant que le tournoi 1 n'est pas entré dans la relation

Tournoi: erreur de référence.

b. Comment corriger cette suite d'instructions pour que le traitement s'effectue sans erreur ?

(1 pt)

Il suffit d'inverser les deux dernières instructions

INSERT INTO ...

3. On donne ci-dessous un extrait des enregistrements contenus dans la base de données :

Tournois		
idtournoi	nom	pays
0	Inconnu	Autre
1	Osaka	Japon
2	Kamakura Games	Japon
3	Meijin	Japon
4	Honinbo	Japon
5	Guksu	Corée
6	Ton Yang Cup	Corée


Joueurs			
idjoueur	nom	naissance	nation
1	Dosaku	1645	Japon
2	Genan Inseki	1798	Japon
3	Shusaku	1829	Japon
4	Kitani Miruno	1909	Japon
5	Go Seigen	1914	Chine
6	Sakata Eio	1920	Japon
7	Rin Kaiho	1942	Taiwan
8	Cho Chikun	1953	Corée
9	Rui Naiwei	1963	Chine
10	Lee Chango	1975	Corée

Parties				
idnoir	idblanc	tournoi	jour	score
2	3	1	1846-09-12	-2
4	5	0	1922-11-12	0
5	4	2	1939-09-28	2
5	6	0	1953-11-19	999
5	4	3	1961-06-28	999
6	5	3	1962-08-05	0
7	6	3	1967-08-09	2
7	8	4	1983-05-16	-999
10	8	6	1993-04-24	0.5
9	10	5	2000-01-04	999

On considère la requête SQL ci-dessous :

```
1 | SELECT COUNT(*) FROM Parties
2 | WHERE idblanc = 4 OR idnoir = 4;
```

- a. Quel serait l'affichage produit par cette requête, appliqué au seuls extraits des enregistrements dans les tableaux précédents ?

 (1 pt) 3

- b. Expliquer avec une phrase ce que renvoie cette requête.

 (1 pt) Il s'agit du nombre de parties jouées par le joueur Kitani Miruno, (d'id 4), avec les noirs ou avec les blancs.

4. Proposer une requête qui renvoie, par ordre alphabétique, les noms des tournois ayant eu lieu au Japon.

 (2 pt)

```
1 | SELECT nom
2 | FROM Tournois
3 | WHERE pays = 'Japon'
4 | ORDER BY pays ASC ;
```

5. Expliquer en une phrase ce que fait la requête ci-dessous :

```
1 | SELECT DISTINCT nom
2 | FROM Joueurs JOIN Parties
3 |           ON (Joueurs.idjoueur = Parties.idnoir
4 |               OR Joueurs.idjoueur = Parties.idblanc)
5 | WHERE Parties.tournoi = 3 ;
```

 (1 pts) Elle affiche, sans doublons, la liste des noms des joueurs qui ont participé au tournoi d'id 3 (Meijin).

6. Proposer une requête qui renvoie les noms des joueurs qui ont joué avec les pierres noires le 15 mars 2016.

 (2 pts)

```
1 | SELECT nom
2 | FROM Joueurs JOIN Parties
3 |           ON Joueurs.idjoueur = Parties.idnoir
4 | WHERE Parties.jour = '2016-03-15' ;
```

### Exercice 3 : Base de données Mozilla (4 points)

Voici le schéma (d'une version simplifiée) de la base de données gérant l'historique (*places*) et les marques pages, ou favoris (*bookmarks*) du navigateur Firefox :

- **moz\_places**(  
    id VARCHAR(50),  
    url VARCHAR(70),  
    title VARCHAR(50),  
    visit\_count INT  
)
- **moz\_bookmarks**(  
    id INT,  
    #fk(moz\_places.id) VARCHAR(50),  
    name VARCHAR(50),  
    #parent(moz\_bookmarks.id) INT,  
    position VARCHAR(20)  
)

Les clés primaires sont soulignées. # indique une clé étrangère (l'attribut référencé est précisé entre parenthèses).

Un marque-page est soit un dossier (son url est alors vide : NULL), soit un marque-page «normal» (avec url non vide, que cette url soit correcte ou non).

1. Pour chaque affirmation, entourer sa valeur booléenne :

- Une même page web peut être enregistrée par deux marques-pages dont les noms sont différents.

☐ vrai ☐ Faux

 (1 pt) Vrai : ces marques-pages peuvent aussi avoir le même nom et le même parent ; leurs id seront différentes.

- Un marque page peut référencer plusieurs pages web.

☐ vrai ☐ Faux

 (1 pt) Faux : l'attribut `fk` renvoie à l'id d'une page : une seule adresse est enregistrée.

2. Expliquer comment modifier ce schéma pour qu'une page ne puisse être référencée que par un seul marque page.

 (2 pts) On peut ajouter une contrainte `UNIQUE` sur l'attribut `fk`.