Évaluation d'expressions mathématiques

1. Introduction

On s'intéresse à l'évaluation d'expressions mathématiques simples contenant des opérandes, des opérateurs, et des fonctions. Par exemple :

$$a + b * (c + d) + \sin(e) \tag{1}$$

a,b,c,d et e sont des opérandes. Il peut s'agir de nombres, de tableaux de nombres, ou de tout type d'objets pour lesquels les opérateurs et les fonctions sont définis. $+$,* sont des opérateurs, qui agissent sur deux opérandes. sin est une fonction, qui agit sur une opérande.

Pour évaluer ce type d'expression, il faut utiliser les règles de calcul associées à cette notation.

2. Définitions

2.a. Priorités des opérateurs

Considérons l'expression suivante :

$$a + b * c \tag{2}$$

La multiplication doit être exécutée avant l'addition, c'est-à-dire que l'on doit sommer a et $b * c$. La multiplication a une priorité supérieure à l'addition. De même, la division est prioritaire sur l'addition (et la soustraction).

2.b. Associativité

Considérons l'expression suivante :

$$a - b + c \tag{3}$$

Elle se calcule comme $(a - b) + c$ et non pas comme $a - (b + c)$. On dit que les opérateur addition et soustraction (qui ont la même priorité) sont associatifs à gauche. De même l'expression :

$$a/b * c \tag{4}$$

se calculer comme $(a/b) * c$ et non pas comme $a/(b * c)$. Les opérateurs multiplication et division sont associatifs à gauche.

L'opérateur de puissance est prioritaire sur la multiplication, et il est associatif à droite avec lui même.

2.c. Notations infixée et suffixée

Comme nous venons de le voir, la notation usuelle des expressions mathématiques fait appel à des règles implicites, auquelles nous sommes habitués. Pour rendre ces règles explicites, il faudrait ajouter des parenthèses dans les expressions. D'autre part, il est souvent nécessaire d'utiliser des parenthèses pour écrire ce type d'expressions.

La notation $a + b$ pour l'addition est appelée notation *infixée*, car le symbole $+$ est placé en infixe par rapport aux opérandes.

Une autre notation consiste à placer l'opérateur en préfixe :

$$+ \ a \ b \tag{5}$$

Cette notation est aussi appelée notation polonaise, en raison de la nationalité du mathématicien qui l'a introduite.

Une notation plus intéressante consiste à placer l'opérateur en suffixe (postfix en anglais) :

$$a \ b \ + \tag{6}$$

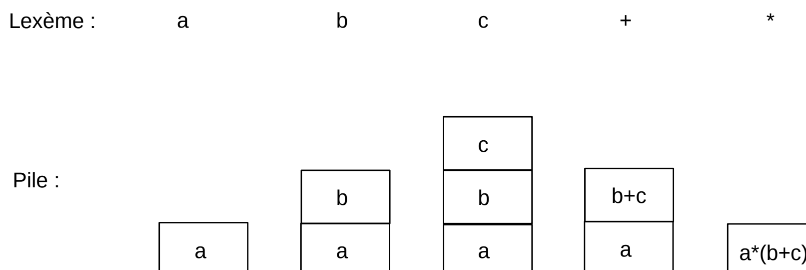
C'est la notation suffixée, appelée aussi notation polonaise inversée. Elle est utilisée dans certaines calculatrices HP, ou dans certains langages informatiques (Forth, Postscript). Cette notation n'est pas très commode pour l'écriture manuscrite, car il faut bien séparer les deux opérandes. En revanche, elle a l'avantage de dispenser complètement des parenthèses. Considérons par exemple l'expression en notation infixée suivante :

$$a * (b + c) \tag{7}$$

En notation suffixée, elle peut s'écrire :

$$a \ b \ c \ + \ * \tag{8}$$

Une expressions suffixée est très facile à évaluer pour un programme informatique. On utilise pour cela une pile contenant les opérandes. L'expression est parcourue de la gauche vers la droite. Lorsqu'une opérande est rencontrée, elle est placée au sommet de la pile. Lorsqu'un opérateur est rencontré, les deux dernières opérandes sont enlevées de la pile, l'opération leur est appliquée, et le résultat est placé au sommet de la pile. Lorsqu'une fonction est rencontrée, la dernière opérande est retirée de la pile, la fonction lui est appliquée, et le résultat est placé au sommet de la pile. Sur les calculatrices HP dont l'écran a plusieurs lignes, la pile est affichée (sommet en bas). Voici les états successifs de la pile (sommet en haut), pour l'évaluation de l'expression $a \ b \ c \ + \ *$:

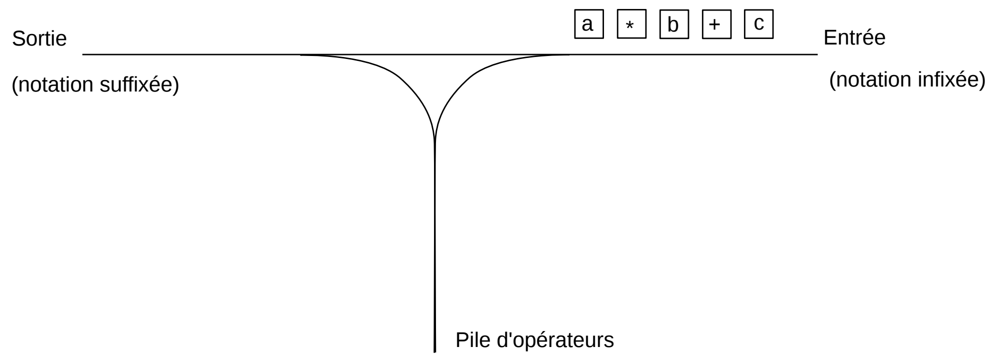


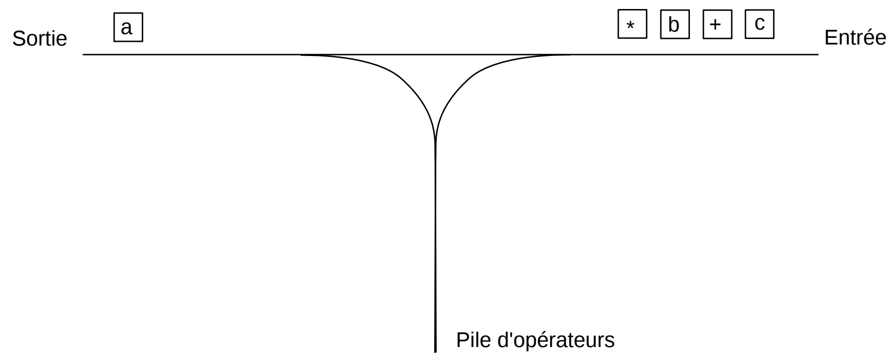
L'évaluation d'une expression suffixée est plus rapide que celle d'une expression infixée, car il n'y a pas de parenthèses à traiter.

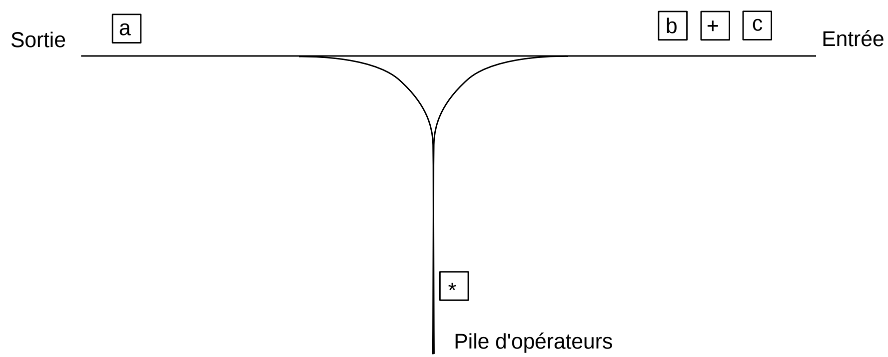
3. Conversion infixée-suffixée

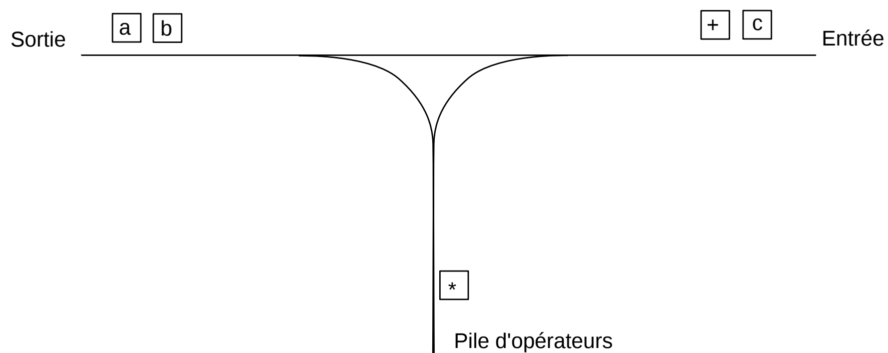
Pour évaluer une expression infixée, une méthode consiste à la convertir tout d'abord en expression suffixée. L'algorithme de la *gare de triage* (shunting yard) permet de faire cette conversion. Les lexèmes de l'expression infixée (l'entrée) sont traités de la gauche vers la droite. Les opérandes sont ajoutés à une liste de sortie. Les opérateurs sont ajoutés dans une pile d'attente. Lorsqu'un opérateur de priorité inférieur à celui du sommet de la pile est traité, celui-ci est enlevé de la pile et placé dans la liste de sortie. Lorsque tous les lexèmes sont traités, la pile est vidée dans la liste de sortie.

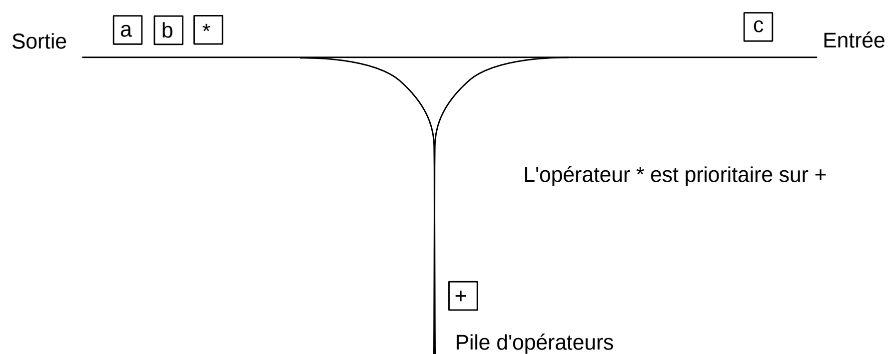
La figure suivante montre le fonctionnement de l'algorithme dans le cas de l'expression $a*b+c$. L'entrée est placée à droite car les lexèmes sont enlevés un par un par la gauche. On imagine un aiguillage qui envoit les opérandes tout droit vers la liste de sortie (à gauche). Les opérateurs sont aiguillées vers la pile, dont le sommet est représenté en haut. Ils sortent de la pile vers la sortie par l'aiguillage de gauche.


a
*
b
+
c
Sortie
Entrée
(notation suffixée)
(notation infixée)
Pile d'opérateurs


Sortie
a
*
b
+
c
Entrée
Pile d'opérateurs


Sortie
a
b
+
c
Entrée
*
Pile d'opérateurs


Sortie
a
b
+
c
Entrée
*
Pile d'opérateurs


Sortie
a
b
*
c
Entrée
L'opérateur * est prioritaire sur +
+
Pile d'opérateurs

L'opérateur + prend la place de l'opérateur * dans la pile car sa priorité est inférieure. Que se passe-t-il lorsque l'opérateur traité a la même priorité que celui du sommet de la pile ? Dans l'expression $a - b + c$, on vérifie facilement que l'opérateur - doit prendre la place de l'opérateur +. S'il ne le fait pas, l'expression obtenue est $a - (b + c)$, qui est fausse. Cela est du au fait que ces deux opérateurs de même priorité sont associatifs à gauche.

Lorsqu'une fonction est rencontrée, elle est ajoutée à la pile d'opérateur. La priorité d'une fonction est supérieure à celle des opérateurs. L'associativité de deux fonctions se fait à droite : lorsqu'une fonction rencontre une autre fonction au sommet de la pile, elle est dirigée directement vers la sortie.

Voyons à présent comment sont traitées les parenthèses. Lorsqu'une parenthèse ouvrante est rencontrée, elle est ajoutée à la pile d'opérateurs. Lorsqu'une parenthèse fermante est rencontrée, il faut vider la pile vers la sortie jusqu'à la première parenthèse ouvrante, laquelle doit bien sûr être enlevée de la pile.

Voici par exemple la séquence pour l'expression $(a - b + c) * d$, représentée dans un tableau. Le haut de la pile d'opérateurs est représenté à droite.

Sortie	*Pile*	*Entre*
	(	a-b+c)*d
a	(	-b+c)*d
a	(-	b+c)*d
a b	(-	+c*d
a b -	(+	c)*d
a b - c	(+	)*d
a b - c +		*d
a b - c +	*	d
a b - c + d	*	
a b - c + d *		

Voici finalement l'algorithme complet décrivant le traitement à appliquer aux lexèmes de l'entrée :

- ▷ Si le lexème est une opérande, l'ajouter à la liste de sortie.
- ▷ Si le lexème est un opérateur, vider la pile vers la sortie tant que la priorité de l'opérateur est inférieure ou égale à celle du lexème du haut de la pile. Si l'associativité de l'opérateur est à droite, la condition inférieure strictement doit être utilisée.
- ▷ Si le lexème est une fonction, le placer dans la pile.
- ▷ Si le lexème est (, le placer dans la pile.
- ▷ Si le lexème est), vider la pile vers la sortie jusqu'à rencontrer). Enlever) de la pile.
- ▷ Lorsque l'entrée est vide, vider la pile dans la sortie.

4. Implémentation en python

```
# -*- coding: utf-8 -*-
import re
```

```
import math
```

4.a. Extraction des lexèmes

On s'intéresse à l'évaluation d'expressions dans lesquelles les opérandes sont des nombres (entiers ou flottants).

On suppose que l'expression est dans une chaîne de caractères. Il s'agit d'extraire les lexèmes (opérandes, opérateurs, etc), de la chaîne. On commence par définir une classe pour représenter les lexèmes en général :

```
class Lexeme:
    def __init__(self,type,valeur):
        self.type = type
        self.valeur = valeur
```

On définit une classe qui hérite de la précédente pour représenter les nombres. On ajoute une fonction `suffixe(pile)`, qui effectuera l'action sur la pile lors de l'évaluation de l'expression suffixée. Dans le cas d'un nombre, il s'agit d'ajouter ce nombre à la pile.

```
class Nombre(Lexeme):
    def __init__(self,valeur):
        Lexeme.__init__(self,"n",valeur)
    def suffixe(self,pile):
        pile.append(self.valeur)
```

La classe suivante représente les opérateurs. On doit ajouter la priorité, sous forme d'un entier, et l'associativité à gauche ou à droite.

```
class Operateur(Lexeme):
    def __init__(self,valeur):
        Lexeme.__init__(self,"op",valeur)
        if valeur=="+":
            self.suffixe = self.plus
            self.priorite = 1
            self.associativite = "g"
        elif valeur=="-":
            self.suffixe = self.moins
            self.priorite = 1
            self.associativite = "g"
        elif valeur=="*":
            self.suffixe = self.mul
            self.priorite = 2
            self.associativite = "g"
        elif valeur=="/":
            self.priorite = 2
            self.suffixe = self.div
            self.associativite = "g"
```

```
        elif valeur=="^":
            self.priorite = 3
            self.suffixe = self.puiss
            self.associativite = "d"

    def plus(self,pile):
        b = pile.pop()
        a = pile.pop()
        pile.append(a+b)
    def moins(self,pile):
        b = pile.pop()
        a = pile.pop()
        pile.append(a-b)
    def mul(self,pile):
        b = pile.pop()
        a = pile.pop()
        pile.append(a*b)
    def div(self,pile):
        b = pile.pop()
        a = pile.pop()
        pile.append(a/b)
    def puiss(self,pile):
        b = pile.pop()
        a = pile.pop()
        pile.append(math.pow(a,b))
```

La classe suivante représente une parenthèse :

```
class Parenthese(Lexeme):
    def __init__(self,valeur):
        Lexeme.__init__(self,"par",valeur)
        self.priorite = 0
```

La classe suivante représente une fonction. Remarquer l'utilisation d'un dictionnaire pour définir la fonction `suffixe(pile)`.

```
class Fonction(Lexeme):
    def __init__(self,valeur):
        Lexeme.__init__(self,"f",valeur)
        suffixe_f = {"sin":self.sin,"cos":self.cos,"exp":self.exp,"log":self.log}
        self.suffixe = suffixe_f[valeur]
        self.priorite = 100
        self.associativite = "d"
    def sin(self,pile):
        a = pile.pop()
        pile.append(math.sin(a))
```

```
    def cos(self,pile):
        a = pile.pop()
        pile.append(math.cos(a))
    def log(self,pile):
        a = pile.pop()
        pile.append(math.log(a))
    def exp(self,pile):
        a = pile.pop()
        pile.append(math.exp(a))
```

L'analyseur syntaxique utilise les expressions régulières pour repérer les lexèmes. La fonction `analyse` prend une chaîne de caractères et renvoit la liste des lexèmes extraits. Elle fonctionne de la manière suivante. La correspondance avec une expression régulière est recherchée sur le début de la chaîne (avec la fonction `re.match`). Si une correspondance est trouvée, la chaîne correspondante est enlevée et traitée par une fonction qui crée le lexème adéquat. Ces fonctions sont définies dans un dictionnaire dont les clés sont les expressions régulières.

```
class AnalyseurSyntaxique:
    def __init__(self):
        self.fonctions_lecture = {"[0-9\.]+":self.lecture_nombre,\
                                  "[+-/*\^]":self.lecture_operateur,\
                                  "[a-z]+":self.lecture_fonction,\
                                  "[\(\)]":self.lecture_parenthese,\
                                  "[\s]+":self.lecture_espace}
        self.lexemes = []
        self.chaine = ""

    def analyse(self,chaine):
        self.lexemes = []
        longueur = len(chaine)
        while longueur > 0:
            for exp in self.fonctions_lecture.keys():
                m = re.match(exp,chaine)
                if m:
                    valeur = chaine[0:m.end()]
                    chaine = chaine[m.end():longueur]
                    longueur -= m.end()-m.start()
                    self.fonctions_lecture[exp](valeur)
                    break
        return self.lexemes

    def lecture_nombre(self,valeur):
        self.lexemes.append(Nombre(float(valeur)))
    def lecture_operateur(self,valeur):
        self.lexemes.append(Operateur(valeur))
    def lecture_parenthese(self,valeur):
```

```
        self.lexemes.append(Parenthese(valeur))
    def lecture_fonction(self,valeur):
        self.lexemes.append(Fonction(valeur))
    def lecture_espace(self,valeur):
        pass
```

4.b. Évaluation d'une expression suffixée

Il suffit pour cela de créer une pile pour placer les opérandes (ici des nombres) et de parcourir la liste des lexèmes en appelant leur fonction `suffixe`.

```
def eval_suffixe(lexemes):
    pile_nombres = []
    for lex in lexemes:
        lex.suffixe(pile_nombres)
    if len(pile_nombres)!=1:
        raise SystemExit("Erreur dans l'expression")
    else:
        return pile_nombres[0]
```

Voici un exemple :

```
analyseur = AnalyseurSyntaxique()
lexemes = analyseur.analyse("2 3 + 5 * 2 -")


print(eval_suffixe(lexemes))
--> 23.0
```

4.c. Conversion infixée-suffixée

La fonction `analyse_infixe(lexemes)` implémente l'algorithme de la gare de triage.

```
def analyse_infixe(lexemes):
    lexemes_sortie = []
    pile_operateurs = []
    for lex in lexemes:
        if lex.type=="n":
            lexemes_sortie.append(lex)
        elif lex.type=="op":
            if len(pile_operateurs)==0:
                pile_operateurs.append(lex)
            else:
                if lex.associativite=="g":
                    while pile_operateurs[-1].priorite >= lex.priorite:
                        lexemes_sortie.append(pile_operateurs.pop())
                        if len(pile_operateurs)==0:
```

```
                        break
            else:
                while pile_operateurs[-1].priorite > lex.priorite:
                    lexemes_sortie.append(pile_operateurs.pop())
                    if len(pile_operateurs)==0:
                        break
            pile_operateurs.append(lex)
        elif lex.type=="par":
            if lex.valeur=="(":
                pile_operateurs.append(lex)
            elif lex.valeur==")":
                while pile_operateurs[-1].valeur!="(":
                    lexemes_sortie.append(pile_operateurs.pop())
                pile_operateurs.pop()
        elif lex.type=="f":
            pile_operateurs.append(lex)
    while len(pile_operateurs)!=0:
        lexemes_sortie.append(pile_operateurs.pop())
    return lexemes_sortie
```

Voici un exemple de conversion :

```
lexemes = analyseur.analyse("2*(3-5)+7")
lexemes = analyse_infixe(lexemes)
valeurs = []
for lex in lexemes:
    valeurs.append(lex.valeur)


print(valeurs)
--> [2.0, 3.0, 5.0, '-', '*', 7.0, '+']

print(eval_suffixe(lexemes))
--> 3.0
```