Évaluation d'une expression en notation polonaise inversée

1. Introduction

La notation polonaise inversée (NPI) permet de définir des expressions mathématiques sans parenthèses. Considérons l'évaluation de l'expression "2+3". En notation polonaise (inventée en 1924 par le mathématicien polonais J. Lukasiewicz), cette expression s'écrirait "+ 2 3". En notation polonaise inversée, on place l'opérateur après les nombres et l'expression s'écrit "2 3 +".

Une expression avec parenthèses, par exemple "2*(5+6)" s'écrit en NPI : "2 5 6 + *".

L'évaluation d'une expression NPI repose sur l'utilisation d'une *pile de calcul*. Dans l'expression précédente, il y a 5 éléments (2,5,6,+,*). L'expression est analysée élément par élément. Lorsqu'on rencontre un nombre, on le place au sommet de la pile. Lorsqu'on rencontre un opérateur binaire (comme + ou *), on enlève les deux derniers nombres de la pile et on place au sommet le résultat de l'opération. D'autres opérateurs, par exemple la fonction *sin*, opèrent sur un seul nombre.

Voici les états successifs de la pile de calcul lors de l'évaluation de l'expression "2 5 6 + *" (le sommet de la pile est à droite) :

```
2
2,5
2,5,6
2,11
22
```

2. Structures de données

La suite des éléments de l'expression est nommée *programme*. On verra en effet qu'il sera possible d'exécuter l'équivalent d'un programme informatique complet.

Le code source du programme est placé dans un fichier texte, chaque ligne du fichier contenant un élément du programme. Les différentes lignes possibles sont :

- ▷ Une ligne débutant par un chiffre représente un nombre, qu'on traitera comme un nombre à virgule flottante.
- ▷ Une ligne débutant par une apostrophe (') contient un nom défini par l'utilisateur (il faudra enlever les apostrophes pour l'obtenir). Un nom est utilisé pour définir une variable.
- ▷ Dans tous les autres cas, la ligne contient le nom d'un opérateur (écrit sans guillemets).

Pour lire le fichier texte, on utilisera le code suivant :

```
fichier = open("programme-0.txt","r")
lignes = fichier.readlines()
for k in range(len(lignes)):
    lignes[k] = lignes[k].strip() # enlève le caractère de fin de ligne
```

qui génère une liste `lignes` contenant les lignes du programme.

La première étape du traitement consiste à parcourir cette liste afin de générer une liste, nommée `programme`, qui contient les éléments suivants :

- ▷ `['nombre',x]` si l'élément est un nombre, avec `x` sa valeur.
- ▷ `['nom',nom]` si l'élément est un nom, avec `nom` ce nom.
- ▷ `['operateur','nom_operateur']` si l'élément est un opérateur.

Voici par exemple le contenu de la liste `programme` pour l'expression donnée en exemple :

```
[['nombre',2],['nombre',5],['nombre',6],['operateur','+'],['operateur','*']]
```

Les variables seront implémentées avec un *dictionnaire* (objet de type `dict`). Un dictionnaire est une structure de données qui permet d'associer une clé à un objet. Les clés sont très souvent des chaînes de caractères. Voici un exemple de dictionnaire :

```
D = {'a':0,'b':1.5}
```

Ce dictionnaire comporte deux clés, les chaînes de caractères `'a'` et `'b'`. La première clé est associée à l'objet 0 (un nombre entier), la seconde est associée à l'objet 1.5 (un flottant). Pour accéder à un objet par sa clé, on utilise la syntaxe suivante :

```
D['a']
```

Les variables (qui contiennent des nombres) seront stockées dans un dictionnaire nommé `variables`. Par exemple, le dictionnaire précédent définit une variable de nom `a` contenant la valeur 0 et une variable de nom `b` contenant le nombre 1.5. Pour ajouter une variable dans le dictionnaire, il suffit d'écrire :

```
D['x'] = 10 # création d'une variable de nom 'x' contenant la valeur 10
```

La création et le rappel d'une variable se fera avec les deux opérateurs suivants :

- ▷ Opérateur `sto` : stocke dans la variable dont le nom se trouve au sommet de la pile le nombre qui se trouve juste avant. Par exemple dans le programme "2 3 'a' sto", le nombre 3 est stocké dans une variable de nom `a`.
- ▷ Opérateur `rcl` : place au sommet de la pile le contenu de la variable dont le nom se trouve au sommet de la pile. Par exemple, dans le programme "2 3 'a' sto 1 'a' rcl", l'opérateur `rcl` place la valeur 3 au sommet de la pile (le nom de la variable est enlevé).

Les opérateurs sont implémentés par des fonctions. La signature d'une fonction d'opérateur est la suivante :

```
def plus(pile,variables): # opérateur plus (+)
```

Une fonction d'opérateur ne renvoie rien, mais elle modifie le contenu de la pile et éventuellement celui des variables (dictionnaire).

Nous avons aussi besoin d'un dictionnaire qui associe à chaque nom d'opérateur (tel qu'il apparaît dans le fichier texte du programme) une fonction Python. Voici un exemplaire de ce dictionnaire :

```
operateurs = {'+':plus,'*':mul,'-':moins,'/':div,'neg':neg,'sto':sto,'rcl':rcl}
```

On notera la présence de l'opérateur `neg`, qui change le nombre situé au sommet de la pile en son opposé (il est impossible d'introduire directement un nombre négatif puisque les nombres commencent par un chiffre).

3. Analyse et exécution du programme

[1] Écrire une fonction `analyse_source(lignes,indice=0)` qui analyse le programme dont les lignes sont dans la liste `lignes`. La fonction `analyse_source` renvoie une liste `programme`. L'analyse de la liste `lignes` se fait à partir de l'indice donné par `indice`, dont la valeur par défaut est 0. On rappelle qu'un paramètre d'une fonction est automatiquement une variable locale de cette fonction. Le parcours de la liste se fera au moyen d'une boucle `while`.

[2] Écrire un programme dans un fichier texte et l'analyser au moyen du code suivant :

```
fichier = open("programme-1.txt","r")
lignes = fichier.readlines()
for k in range(len(lignes)):
    lignes[k] = lignes[k].strip()
programme = analyse_source(lignes)
print(programme)
```

[3] Écrire les fonctions `plus,mul,neg,div,moins,sto,rcl`, qui implémentent les opérateurs. Définir le dictionnaire `operateurs`.

[4] Écrire une fonction `execution(programme,pile,variables,operateurs)` qui exécute le programme donné dans la liste `programme`. Cette fonction modifie le contenu de la pile et des variables. On exécutera donc le programme de la manière suivante :

```
pile = []
variables = {}
execution(programme,pile,variables,operateurs)
print(pile)
print(variables)
```

[5] Tester un programme comportant l'évaluation d'une expression, avec définition d'une variable et rappel de sa valeur. On remarquera qu'il est possible d'évaluer plusieurs expressions dans le même programme. La pile contient finalement les résultats des évaluations.

4. Sous-programme

Un sous-programme, qu'on nommera *routine*, est un programme qui est stocké en mémoire afin d'être utilisé plusieurs fois. Un nom lui est associé. C'est donc l'équivalent d'une fonction en Python. Pour définir une routine, on placera dans le code source (le fichier texte) une ligne contenant { et la routine se termine par une ligne contenant }. Cette ligne devra être suivie du nom que l'on veut attribuer à la routine, sous la forme `'nom'` (avec deux apostrophes).

Voici par exemple un programme qui contient la définition d'une routine nommée `double`, qui multiplie un nombre par deux. La routine est ensuite appliquée au nombre 10.

```
{
2
*
}
'double'
10
double
```

Une fois la routine définie, son appel se fait comme celui d'un opérateur. Bien sûr, on ne doit pas attribuer à une routine le nom d'un opérateur existant.

Les routines seront stockées dans un dictionnaire. Voici par exemple le dictionnaire qui contiendrait la routine précédente :

```
routines = {'double':[['nombre',2],['operateur','*']]}
```

Les variables définies dans le dictionnaire `variables` seront accessibles aussi bien depuis le programme principal que depuis une routine. On n'implémentera pas de variables locales pour les routines.

L'analyse d'un code source de programme contenant des routines se fait aisément au moyen de la récursivité. Il en est de même de son exécution.

[6] Dans un nouveau script, écrire une fonction `analyse_source(lignes,routines,indice=0)` qui prend en charge la définition des routines.

[7] Écrire une fonction `execution(programme,pile,variables,operateurs,routines)` qui exécute un programme en prennant en charge l'exécution d'une routine lorsqu'un nom faisant partie des clés de `routines` est rencontré. La liste des clés d'un dictionnaire est obtenu de la manière suivante :

```
liste_cles = routines.keys() # liste des noms de routines déjà définies
```

5. Solution

5.a. Programme

evaluation-npi.py

```
fichier = open("programme-0.txt","r")
lignes = fichier.readlines()
for k in range(len(lignes)):
    lignes[k] = lignes[k].strip()



def analyse_source(lignes,indice=0):
    programme = []
    while indice < len(lignes):
        ligne = lignes[indice]
        if ligne[0] in ['0','1','2','3','4','5','6','7','8','9']:
```

```
            programme.append(['nombre',float(ligne)])
        elif ligne[0]=="'":
            ligne = ligne.replace("'","")
            programme.append(["nom",ligne])
        else:
            programme.append(["operateur",ligne])
        indice += 1
    return programme

programme = analyse_source(lignes)
print(programme)

pile=[]
variables = {}

def plus(pile,variables):
    x = pile.pop()
    y = pile.pop()
    pile.append(x+y)

def mul(pile,variables):
    x = pile.pop()
    y = pile.pop()
    pile.append(x*y)

def neg(pile,variables):
    x = pile.pop()
    pile.append(-x)

def sto(pile,variables):
    nom = pile.pop()
    x = pile.pop()
    variables[nom] = x

def rcl(pile,variables):
    nom = pile.pop()
    pile.append(variables[nom])

operateurs = {'+':plus,'*':mul,'neg':neg,'sto':sto,'rcl':rcl}

def execution(programme,pile,variables,operateurs):
    for p in programme:
        if p[0]=='nombre':
            pile.append(p[1])
        elif p[0]=='nom':
            pile.append(p[1])
        elif p[0]=='operateur':
            f = operateurs[p[1]]
            f(pile,variables)

execution(programme,pile,variables,operateurs)
print(pile)
```

5.b. Programme avec sous-programmes

evaluation-routines-npi.py

```
fichier = open("programme.txt","r")
lignes = fichier.readlines()
for k in range(len(lignes)):
    lignes[k] = lignes[k].strip()
programme = []
routines = {}

def analyse_source(lignes,routines,indice=0):
    programme = []
    while indice < len(lignes):
        ligne = lignes[indice]

        if ligne[0] in ['0','1','2','3','4','5','6','7','8','9']:
            programme.append(['nombre',float(ligne)])
        elif ligne[0]=="'":
            ligne = ligne.replace("'","")
            programme.append(["nom",ligne])
        elif ligne[0] == "{": # début sous-programme
            lignes_routine = []
            indice += 1
            while lignes[indice]!="}":
                lignes_routine.append(lignes[indice])
                indice += 1
            routine = analyse_source(lignes_routine,routines)
            indice += 1
            nom = lignes[indice].replace("'","")
            routines[nom] = routine
        else:
            programme.append(["operateur",ligne])
        indice += 1
    return programme

programme = analyse_source(lignes,routines)

print(programme)
print(routines)

pile=[]
variables = {}

def plus(pile,variables):
    x = pile.pop()
    y = pile.pop()
    pile.append(x+y)

def mul(pile,variables):
    x = pile.pop()
    y = pile.pop()
    pile.append(x*y)

def neg(pile,variables):
    x = pile.pop()
    pile.append(-x)

def sto(pile,variables):
    nom = pile.pop()
    x = pile.pop()
```

```
    variables[nom] = x

def rcl(pile,variables):
    nom = pile.pop()
    pile.append(variables[nom])

operateurs = {'+':plus,'*':mul,'neg':neg,'sto':sto,'rcl':rcl}

def execution(programme,pile,variables,operateurs,routines):
    for p in programme:
        if p[0]=='nombre':
            pile.append(p[1])
        elif p[0]=='nom':
            pile.append(p[1])
        elif p[0]=='operateur':
            if p[1] in routines.keys():
                execution(routines[p[1]],pile,variables,operateurs,routines)
            else:
                f = operateurs[p[1]]
                f(pile,variables)

execution(programme,pile,variables,operateurs,routines)
print(pile)
```