tp_l8dn003_20_06_cor

April 6, 2020

TP n°6: analyse expressions arithmétiques en notation polonaise inverse

1. Implémentation d'une pile

Proposer une implémentation au moyen d'une liste des quatre primitives indispensables pour manipuler une pile: cree_pile(), vide(), empiler(), depiler(). La définition fonctionnelle des primitives peut être déduite des exemples suivants:

```
print(depiler(empiler(empiler(empiler(cree_pile(),3),5),7)))
7
print(vide(empiler(cree_pile(),5)))
False
```

2. Evaluation d'une formule en notation polonaise inverse

Ecrire un programme qui prend en entrée une expression arithmétique en notation polonaise inverse (sous la forme d'une chaîne de caractère), et qui donne en sortie la valeur de l'expression. L'algorithme utilisé repose sur l'idée que chaque opérande rencontré est empilé, et que chaque fois qu'on rencontre un opérateur, on dépile deux arguments (les deux derniers, donc), on calcule le résultat, et on l'empile. Si l'expression est bien formée, la valeur de l'expression se trouve au sommet de la pile à la fin du calcul.

```
3 12 3 - + = 12
7 7 * = 49
23 12 - 8 + 2 * = 38
```

Il n'est pas demandé de vérifier que l'expression est bien formée (mais on peut essayer d'ajouter une vérification pour éviter les erreurs). Noter que si l'on veut ajouter des symboles unaires (le signe "moins" dans l'expression "-3") on doit utiliser des symboles différents de leur correspondant binaire.

3. Traduction d'une formule arithmétique

Ecrire un programme qui prend en entrée une expression arithmétique en notation polonaise inverse, et donne en sortie une chaîne de caractère correspondant à l'expression équivalente en notation infixe. La pile définie plus haut devrait pouvoir être utilisée.

```
3 12 3 - + = (3 + (12 - 3))
7 7 * = (7 * 7)
23 12 - 8 + 2 * = (((23 - 12) + 8) * 2)
```

1. Implémentation d'une pile

On propose une version simple (non objet) des quatre primitives. Il n'y a pas de contrôle d'erreur (par exemple on pourrait vérifier que la pile est bien définie, que c'est une liste, que la valeur est bien du bon type, et renvoyer un message d'erreur si on tente de dépiler une pile vide).

On propose ensuite une version "objet", qui permet une manipulation un peu plus aisée, mais qui devrait aussi comprendre des contrôles pour éviter les cas d'erreur.

[1]:
```
def cree_pile():
    return []

def vide(p):
    return p == [] # or p == None

def empiler(p,val):
    p.append(val)
    return p

def depiler(p):
    if not vide(p):
        return p.pop()
```

[2]:
```
# Quelques tests des primitives
print(depiler(empiler(empiler(empiler(cree_pile(),3),5),7)))
print(vide(empiler(cree_pile(),5)))
```

```
7
False
```

[3]:
```
# version "objet" de la pile (minimale)
# Aucun contrôle d'erreur dans les méthodes
class maPile:
    def __init__(self):
        self._l = []

    def empile(self, val):
        self._l.append(val)

    def depile(self):
        return self._l.pop()

    def vide(self):
        return len(self._l) == 0
```

2. Interprétation d'une formule en NPI

On propose deux fonctions différentes qui utilisent respectivement le premier jeu de primitive et la version objet.

[4]:
```
# Interprétation d'une expression en NPI
def interpretation(expr):
    p = cree_pile()
    for token in expr.split():
        if token == '+':
            empiler(p,depiler(p)+depiler(p))
        elif token == '*':
            empiler(p,depiler(p)*depiler(p))
        elif token == '-':
            op2 = depiler(p)
            op1 = depiler(p)
            empiler(p, op1 - op2)
        else:
            empiler(p,int(token))
    return depiler(p)
```

[5]:
```
def interpretation_oo(expr):
    p = maPile()
    for token in expr.split():
        if token == '+':
            p.empile(p.depile()+p.depile())
        elif token == '*':
            p.empile(p.depile()*p.depile())
        elif token == '-':
            op2 = p.depile()
            op1 = p.depile()
            p.empile(op1 - op2)
        else:
            p.empile(int(token))
    return p.depile()
```

[6]:
```
le = ['3 12 3 - +', '7 7 *', '23 12 - 8 + 2 *']
for e in le:
    print("%s = %d" % (e,interpretation(e)))
```

```
3 12 3 - + = 12
7 7 * = 49
23 12 - 8 + 2 * = 38
```

[7]:
```
le = ['3 12 3 - +', '7 7 *', '23 12 - 8 + 2 *']
for e in le:
    print("%s = %d" % (e,interpretation_oo(e)))
```

```
3 12 3 - + = 12
7 7 * = 49
23 12 - 8 + 2 * = 38
```

3. Traduction d'une expression en NPI vers le format infixe parenthésé.

On utilise cette fois la pile pour stocker des chaînes de caractères, l'algorithme est sensiblement le même que pour l'interprétation, la pile jouant encore une fois un rôle majeur.

Ici on exploite le fait que nous n'avons pas défini le type des valeurs qu'on empile sur la pile: on l'avait utilisée pour des entiers, mais il est possible de l'utiliser pour des str. Avec un contrôle de type plus strict, on aurait pu être amené à définir deux classes différentes, les piles d'entiers, et les piles de str.

[8]:
```
def traduction(expr):
    trad = ""
    p = cree_pile()  # la pile telle qu'elle est définie peut stocker des str
    for token in expr.split():
        if token in ['+', '-', '*', '/']:
            op2 = depiler(p)
            op1 = depiler(p)
            empiler(p, "(%s %s %s)" % (op1,token,op2))
        else:
            empiler(p,token)
    return depiler(p)
```

[9]:
```
le = ['3 12 3 - +', '7 7 *', '23 12 - 8 + 2 *']
for e in le:
    print("%s = %s" % (e,traduction(e)))
```

```
3 12 3 - + = (3 + (12 - 3))
7 7 * = (7 * 7)
23 12 - 8 + 2 * = (((23 - 12) + 8) * 2)
```