DS 2 d'informatique de l'année 2022/2023 - MPSI

Calculatrices interdites. Pensez à numéroter vos feuilles.
Sur votre copie, les questions doivent apparaître dans l'ordre de l'énoncé.

Le but de ce sujet est d'écrire une fonction « `executer(prog: str) -> NoneType` » qui exécute des opérations arithmétiques simples contenues dans une chaîne de caractères `prog`. Par exemple le programme suivant affiche 98 (ligne 4), puis 3 (ligne 6), puis -45 (ligne 11).

```
1  s = """
2  # Ceci est un commentaire
3  # Les commentaires seront supprimés avant l'exécution
4  98       # Cette ligne affiche 98
5  a = 9+6  # Variable a qui vaut 15
6  a - 12
7
8  b = -4+a
9  c = 8+a
10 a = -a + 10
11 -b+a+1 - (7+c)
12 """
13 executer(s)
```

Rappels concernant les chaînes de caractères.

- ⋆ La longueur d'une chaîne de caractères `s` est donnée par la commande `len(s)`.
- ⋆ On appelle "caractère" une chaîne de caractères de longueur 1, par exemple le retour chariot `"\n"` est un caractère permettant de passer à la ligne (en particulier `len("\n")` s'évalue en 1).
- ⋆ Pour tout `i` $\in [\![$ `0; len(s) - 1` $]\!]$, la commande `s[i]` permet d'accéder au caractère d'indice `i` de `s`.
- ⋆ Si `s1` et `s2` sont deux chaînes de caractères, alors l'expression `s1 + s2` désigne la concaténation de `s1` et `s2`.
- ⋆ La méthode `.append` ne fonctionne pas pour les chaînes de caractères.
- ⋆ Pour `0` $\leqslant$ `i` $\leqslant$ `j` $\leqslant$ `len(s)`, l'expression `s[i:j]` est la chaîne de caractères composée des caractères `s[i], s[i+1], ..., s[j-1]`. En particulier, si `i = j`, alors `s[i:j]` est la chaîne de caractères vide.
- ⋆ Les méthodes `split` et `join` sont interdites.

1 Préliminaires

Les fonctions écrites dans cette partie pourront être réutilisées librement.

1. (a) Écrire une fonction `indRC(prog: str) -> list[int]` qui renvoie une liste `L` contenant tous les indices `i` tels que `prog[i]` est un retour chariot. Par exemple, pour `progQ1`, on obtient `LQ1`.
   (b) En déduire une fonction `separer(prog: str) -> list[str]` qui renvoie une liste `M` dont les éléments sont les différentes lignes de `prog`. Par exemple, pour `progQ1`, on obtient `MQ1`. Notez qu'en particulier, les chaînes de caractères contenues dans `M` ne contiennent pas de retour chariot.
2. À l'aide d'une compréhension de liste, écrire une fonction `supprVides(M: list[str]) -> list[str]` qui renvoie une liste contenant tous les éléments de `M` sauf les chaînes de caractères vides. Par exemple, à partir de `MQ1`, on obtient `MQ2`. Le corps de votre fonction devra contenir uniquement un `return` et une compréhension de liste.

```
progQ1 = """
Ex:
1ère ligne
2ème ligne\n3ème ligne

..."""
```

```
LQ1 = [0, 4, 15, 26, 37, 38]
```

```
MQ1 = ['', 'Ex:', '1ère ligne',
       '2ème ligne', '3ème ligne', '', '...']
```

```
MQ2 = ['Ex:', '1ère ligne', '2ème ligne', '3ème ligne',
   '...']
```

s	"Ceci est un test !"	"a = 1+1 # Commentaire"
supprSPC(s)	"Ceciestuntest!"	"a=1+1#Commentaire"
supprCom(s)	"Ceci est un test !"	"a = 1+1 "

3. Écrire une fonction `supprSPC(s: str) -> str` qui renvoie une chaîne de caractères contenant les mêmes caractères que `s` sans les espaces. Voir les exemples ci-dessus.
4. À l'aide d'une boucle `while`, écrire une fonction `supprCom(s: str) -> str` qui supprime les commentaires d'une chaîne de caractères. On pourra supposer sans le vérifier que `s` ne contient pas de retour chariot. Voir les exemples ci-dessus.

À partir de maintenant et sauf dans les questions 14, on manipulera uniquement des chaînes de caractères ne contenant aucun retour chariot, aucun espace et aucun commentaire.

5. (a) Écrire une fonction `strToDict(s: str) -> dict[str, list[int]]` qui renvoie un dictionnaire associant à chaque caractères `c` de `s`, une liste contenant les indices d'apparitions de `c` dans `s`. Votre fonction devra s'exécuter en temps linéaire en `len(s)`. Par exemple, pour la chaîne de caractères `"Ceciestuntest!"`, on obtient :

```
{'C': [0], 'e': [1, 4, 10], 'c': [2], 'i': [3],
 's': [5, 11], 't': [6, 9, 12], 'u': [7], 'n': [8], '!': [13]}
```

   (b) Montrer que le temps d'exécution de votre fonction est bien linéaire en `len(s)`.

Étant donnée `(s: str)`, on souhaite vérifier si tous les caractères de `s` appartiennent à un certain ensemble C. En Python, un ensemble C est représenté par un dictionnaire de type `dict[str, NoneType]` dont les clés sont les éléments de C et dont toutes les valeurs sont égales à `None`. Par exemple, l'ensemble des chiffres $C = \{$`"0"`, `"1"`, `"2"`, `"3"`, `"4"`, `"5"`, `"6"`, `"7"`, `"8"`, `"9"`$\}$ correspond au dictionnaire `C_chiffres` :

```
C_chiffres = {"0": None, "1": None, "2": None, "3": None, "4": None,
              "5": None, "6": None, "7": None, "8": None, "9": None}
```

De même, l'ensemble des lettres majuscules et minuscules de l'alphabet est représenté par :

```
C_lettres = {
    "a":None, "b":None, "c":None, "d":None, "e":None, "f":None, "g":None, "h":None,
    "i":None, "j":None, "k":None, "l":None, "m":None, "n":None, "o":None, "p":None,
    "q":None, "r":None, "s":None, "t":None, "u":None, "v":None, "w":None, "x":None,
    "y":None, "z":None, "A":None, "B":None, "C":None, "D":None, "E":None, "F":None,
    "G":None, "H":None, "I":None, "J":None, "K":None, "L":None, "M":None, "N":None,
    "O":None, "P":None, "Q":None, "R":None, "S":None, "T":None, "U":None, "V":None,
    "W":None, "X":None, "Y":None, "Z":None}
```

6. (a) Écrire une fonction `verifCarac(s: str, C: dict[str, NoneType]) -> bool` qui indique si `s` est non vide et contient uniquement des caractères présents dans `C`. Par exemple, `verifCarac(s, C_chiffres)` vaut `True` pour `s = "159648"` et `False` pour `s = "159+203"`.
   (b) Quelle est la complexité de votre fonction ?

Jusqu'à la fin du sujet, on suppose que les dictionnaires `C_chiffres` et `C_lettres` sont définis comme des variables globales. On pourra donc les utiliser même s'ils ne sont pas donnés en argument des fonctions.

2 Évaluation d'expressions arithmétiques

Soit $\mathbb{V}$ l'ensemble des chaînes de caractères `s` telles que `verifCarac(s, C_lettres)` s'évalue en `True` (c'est à dire que `s` est non vide et ne contient que des lettres). Un élément de $\mathbb{V}$ est appelé un "nom de variable". On note également $\mathbb{A}$ l'ensemble des chaînes de caractères contenant une expression arithmétique formée avec les opérateurs `+` et `-`, des entiers positifs ou négatifs, des noms de variables et des parenthèses (les espaces et retours à la ligne ne sont pas autorisés). Par exemple, voici quatre éléments de $\mathbb{A}$:

```
s1 = "0"
s2 = "-34+8"
s3 = "x+y-15+nomVariable"
s4 = "(-4+x)-(17-(x+(-y)+6))"
```

Conversions entre entiers et chaînes de caractères.

7. (a) Donner la signature et la spécification de la fonction `f0` :

```
def f0(x):
    d = {"0": 0, "1": 1, "2": 2, "3": 3, "4": 4,
         "5": 5, "6": 6, "7": 7, "8": 8, "9": 9}
    assert x in d
    return d[x]
```

   (b) Soit `C_chiffres` le dictionnaire défini dans la partie 1 et `s` une chaîne de caractères telle que `verifCarac(s, C_chiffres)` vaut `True` (c'est à dire que `s` est non vide et ne contient que des chiffres). À l'aide d'une boucle et de la fonction `f0`, écrire une fonction `strToInt(s: str) -> int` qui renvoie l'entier contenu dans `s`. En d'autres termes, votre fonction doit renvoyer `int(s)`, mais sans utiliser la fonction `int`. Par exemple, `strToInt("156")` est l'entier `156`. Dans le cas où `verifCarac(s, C_chiffres)` ne vaut pas `True`, votre fonction déclenchera une erreur (rappel : `C_chiffres` est une variable globale).

8. À l'aide d'une boucle, écrire une fonction `intToStr(n: int) -> str` qui renvoie la chaîne de caractères correspondant à `n`. En d'autres termes, votre fonction doit renvoyer `str(n)`, mais sans utiliser la fonction `str`. Par exemple, `intToStr(156)` vaut `"156"` et `intToStr(-156)` vaut `"-156"`.

Expressions arithmétiques bien parenthésées. Soit $s \in \mathbb{A}$. On dit que s est ***bien parenthésée*** si les deux conditions suivantes sont vérifiées :

- Le nombre de parenthèses ouvrantes est égal au nombre de parenthèses fermantes.
- Pour tout caractère c, le nombre de parenthèses ouvrantes qui précèdent c est supérieur ou égal au nombre de parenthèses fermantes qui précèdent c.

Par exemple, `"(x+y)-(z+2"` n'est pas bien parenthésée à cause la première condition et `"(x+y)-(z+2))("` n'est pas bien parenthésée à cause de la seconde condition appliquée au caractère d'indice 11.

9. Écrire une fonction `bienParenthesee(s: str) -> bool` qui indique si `s` est bien parenthésée.

Expressions arithmétiques normalisées. Un opérateur `+` ou `-` peut être appliqué à un ou deux arguments. Par exemple :

- L'opérateur `-` est appliqué à deux arguments dans l'expression `y-15` de `s3`. On parle alors ***d'opérateur binaire***.
- L'opérateur `-` est appliqué à un seul argument dans les expressions `-34` de `s2` et `-y` de `s4`. On parle alors ***d'opérateur unaire***.

Soit $s \in \mathbb{A}$. On dit que s est ***normalisée*** si s ne contient ni parenthèse, ni opérateur unaire. Par exemple `s1` et `s3` sont normalisées, mais `s2` et `s4` ne le sont pas. De plus, les expressions normalisées `r2` et `r4` sont équivalentes à `s2` et `s4` :

```
r2 = "0-34+8"
r4 = "0-4+x-17+x-y+6"
```

10. Écrire une fonction `normaliser(s: str) -> str` qui prend en entrée un élément de $\mathbb{A}$ et renvoie une chaîne de caractères normalisée équivalente à `s`. Si `bienParenthesee(s)` vaut `False` votre fonction déclenchera une erreur. On pourra supposer sans le vérifier que `s` appartient à $\mathbb{A}$. On expliquera succinctement la procédure utilisée.

Évaluation d'expressions arithmétiques normalisées. Soit $s \in \mathbb{A}$ une expression arithmétique normalisée dont on souhaite déterminer la valeur. Puisque s peut contenir des noms de variables, il faut pouvoir accéder aux valeurs de ces variables. Pour cela, on considère un dictionnaire `(dVar: dict[str, int])` tel que chaque clé `c` est le nom d'une variable et `dVar[c]` est la valeur associée. Par exemple, si `dVar` vaut :

```
dVar = {"x": -7, "y": 6, "nomVariable": 3, "variableInutilisee": 42}
```

alors l'expression `s3` a pour valeur $-7+6-15+3=-13$.

11. (a) Écrire une fonction `decomposer(t: str) -> (list[str], list[str])` qui prend en entrée une chaîne de caractères `t` et renvoie deux listes `(L1, L2)` telles que `L1` contient les opérateurs `"+"` et `"-"` présents dans `t` et :

```
t = L2[0] + L1[0] + L2[1] + L1[1] + ... + L2[n-2] + L1[n-2] + L2[n-1]
```

où `n = len(L2)`. Par exemple :

t	"0"	"x+y-15+nomVariable"	"-34+8"
L1	[]	["+", "-", "+"]	["-", "+"]
L2	["0"]	["x", "y", "15", "nomVariable"]	["", "34", "8"]

(b) Écrire une fonction `estExprNorm(s: str, dVar: dict[str, int]) -> bool` qui vérifie si l'expression `s` est normalisée et si toutes ses opérandes (c'est à dire les arguments des opérateurs) correspondent bien à des entiers positifs ou des noms de variables de `dVar`. Lorsque ces conditions sont respectées, la fonction renvoie `True`, sinon elle renvoie `False`. On justifiera succinctement la procédure utilisée.

(c) Écrire une fonction `evalExprNorm` qui prend en entrée une expression arithmétique normalisée $s \in \mathbb{A}$ ainsi que `dVar`, et renvoie la valeur de s. Si `estExprNorm(s, dVar)` vaut `False` votre fonction déclenchera une erreur.

Évaluation d'expressions arithmétiques quelconques. Dans cette partie, $s \in \mathbb{A}$ est une expression arithmétique (pas nécessairement normalisée) pouvant contenir des parenthèses (les espaces, retours à la ligne et commentaires sont toujours interdits).

12. Écrire une fonction `evalExpr` qui prend en entrée une expression arithmétique $s \in \mathbb{A}$ ainsi que `dVar`, et renvoie la valeur de s. On pourra supposer sans le vérifier que `s` est bien une expression arithmétique.

3 Affectation de variables

Soit $\mathbb{B}$ l'ensemble des chaînes de caractères `s` pouvant être décomposées sous la forme :

```
s = s1 + "=" + s2
```

où `s1` est un nom de variable et `s2` $\in \mathbb{A}$ est une expression arithmétique. Par exemple la chaîne `"x=-x-(4+y)-2"` appartient à $\mathbb{B}$.

13. Écrire une fonction `affectation(s: str, dVar: dict[str, int]) -> NoneType` qui prend en entrée `s` $\in \mathbb{B}$ et modifie le dictionnaire `dVar` pour prendre en compte l'affectation `s`. On pourra supposer sans le vérifier que `s` $\in \mathbb{B}$.

14. Écrire la fonction `executer(prog: str) -> NoneType` décrite au début de l'énoncé.