DS 2 d'informatique - MPSI - Évaluation d'expressions arithmétiques - Correction

Question 1.a –

```
def indRC(s):
    """indRC(s: str) -> list[int]"""
    L = []
    for i in range(len(s)):
        if s[i] == '\n':
            L.append(i)
    return L
```

Question 1.b –

```
def separer(s):
    """separer(s: str) -> list[str]"""
    L = indRC(s)
    if len(L) == 0:
        return [s]
    M = [s[0:L[0]]]
    for i in range(len(L)-1):
        M.append(s[L[i]+1 : L[i+1]])
    M.append(s[L[-1]+1:len(s)])
    return M
```

Question 2 –

```
def supprVides(M):
    """supprVides(M: list[str]) -> list[str]"""
    return [s for s in M if s != ""]
```

Question 3 –

```
def supprSPC(s):
    """supprSPC(s: str) -> str"""
    t = ""
    for i in range(len(s)):
        if s[i] != ' ':
            t = t+s[i]
    return t
```

Question 4 –

```
def supprCom(s):
    """supprCom(s: str) -> str"""
    t = ""
    i = 0
    while i < len(s) and s[i] != '#':
        t = t+s[i]
        i = i+1
    return t
```

Question 5.a –

```
def strToDict(s):
    """strToDict(s: str) -> dict[str, list[int]]"""
    d = {}
    for i in range(len(s)):
        c = s[i]
        if c in d:
            d[c].append(i)
        else:
            d[c] = [i]
    return d
```

Question 5.b – Le nombre de tour de boucle est `len(s)` et chaque tour de boucle s'exécute en temps constant (en particulier le test « `c in d` » s'exécute en temps constant). Ainsi le temps d'exécution est linéaire en `len(s)`.

Question 6.a –

```
def verifCarac(s, C):
    """verifCarac(s: str, C: dict[str, None]) -> bool"""
    if s == "":
        return False
    for c in s:
        if c not in C:
            return False
    return True
```

Question 6.b – La fonction s'exécute en temps linéaire en `len(s)`. En effet, il y a `len(s)` tours de boucle et chaque tour s'exécute en temps constant.

Question 7.a – La signature de `f0` est :

`f0(x: str) -> int`.

Cette fonction prend entrée une chaîne de caractères contenant un unique chiffre et renvoie l'entier correspondant (par exemple `f0("4")` s'évalue en `4`). Si la chaîne de caractères ne contient pas un unique chiffre, la fonction déclenche une erreur.

Question 7.b –

```
def strToInt(s):
    """strToInt(s: str) -> int"""
    assert verifCarac(s, C_chiffres)
    res = 0
    for c in s:
        res = res*10 + f0(c)
    return res
```

Question 8 –

```
# Suppose n > 0
# On aurait pu créer une liste à la place de d.
def intPosToStr(n):
    """intPosToStr(n: int) -> str"""
    d = {0: "0", 1: "1", 2: "2", 3: "3", 4: "4",
         5: "5", 6: "6", 7: "7", 8: "8", 9: "9"}
    s = ""
    while n > 0:
        s = d[n%10] + s
        n = n//10
    return s
```

```
def intToStr(n):
    """intToStr(n: int) -> str"""
    if n == 0:
        return "0"
    elif n > 0:
        return intPosToStr(n)
    else:
        return "-" + intPosToStr(-n)
```

Question 9 –

```
def bienParenthesee(s):
    """bienParenthesee(s: str) -> bool"""
    nb1 = 0
    nb2 = 0
    for c in s:
        if c == "(":
            nb1 += 1
        elif c == ")":
            nb2 += 1
        if nb2 > nb1:
            return False
    return nb1 == nb2
```

Question 10 – On supprime les parenthèses de `s` petit à petit. Pour cela :

- On repère un couple de parenthèses qui ne contient pas d'autre parenthèse (avec la fonction `indParentheses`).
- Pour supprimer ce couple de parenthèses, on distingue les cas où le caractère qui précède est un `"+"` ou `"-"`.
- On recommence tant que `s` contient des parenthèses.

Lorsque toutes les parenthèses sont supprimées, on ajoute 0 au début de `s` si elle commence par l'opérateur unaire `"-"` et on supprime le premier caractère de `s` si c'est `"+"`.

Avant de supprimer les parenthèses, la chaîne de caractères `s` subit un pré-traitement : on fait en sorte que `s` commence par un opérateur unaire et que chaque parenthèse ouvrante soit suivie d'un opérateur unaire. De plus, les enchaînements d'opérateurs sont simplifiés (`--1` est simplifié en `+1`). Pour cela :

- La fonction `ajoutUnaires` ajoute un `+` au début de la chaîne de caractères et après les parenthèses ouvrantes.
- La fonction `simplifierUnaires` simplifie les enchaînements d'opérateurs.

```
# Ajoute des + après toutes les parenthèses ouvrantes et au début de la str
def ajoutUnaires(s):
    """ajoutUnaires(s: str) -> str"""
    t = "+"
    for c in s:
        t += c
        if c == '(':
            t += '+'
    return t
```

```
# Suppose s non vide
# Simplifie les enchaînements d'opérateurs unaires
def simplifierUnaires(s):
    """simplifierUnaires(s: str) -> str"""
    t = s[0]
    d = {("+","+"): "+", ("+","-"): "-", ("-","+"): "-", ("-","-"): "+"}
    for i in range(1,len(s)):
        k = (t[-1], s[i])
        if k not in d:
            t += s[i]
        else:
            t = t[:-1] + d[k]
    return t
```

```
# Renvoie les indices d'un couple de parenthèses qui ne contiennent pas d'autre
# parenthèse.
# Renvoie None si s ne contient pas de parenthèse.
# Hypothèse: s est bien parenthésée
def indParentheses(s):
    """indParentheses(s: str) -> (int, int) ou NoneType"""
    i = None
    for j in range(len(s)):
        if s[j] == ')':
            return i, j
        elif s[j] == '(':
            i = j
    return None
```

```
# Cette fonction remplace + par - et inversement.
def changerSigne(s):
    """changerSigne(s: str) -> str"""
    d = {"+": "-", "-": "+"}
    t = ""
    for c in s:
        if c in d:
            t += d[c]
        else:
            t += c
    return t
```

```
# Suppose que s a subi les prétraitements ajoutUnaire et SimplifierUnaires
def supprParentheses(s):
    """supprParentheses(s: str) -> str"""
    while True:
        t = indParentheses(s)
        if t is None:  # OU t == None
            break
        i,j = t
        if s[i-1] == '-':
            s = s[:i-1] + changerSigne(s[i+1:j]) + s[j+1:]
        else:  # s[i-1] == '+'
            s = s[:i-1] + s[i+1:j] + s[j+1:]
    return s
```

```
# Suppose s non vide
def normaliser(s):
    """normaliser(s: str) -> str"""
    assert bienParenthesee(s)
    s = ajoutUnaires(s)
    s = simplifierUnaires(s)
    s = supprParentheses(s)
    if s[0] == "-":
        return "0" + s
    else: # s[0] == "+":
        return s[1:]
```

Remarque. Si `s` n'est pas une expression arithmétique, le comportement de `normaliser` peut être inattendu. Par exemple, pour `s = "(1)1"`, on obtient `"11"`.

Question 11.a –

```
def decomposer(s):
    """decomposer(s: str) -> (list[str], list[str])"""
    L1 = []
    L2 = [""]
    for c in s:
        if c == '+' or c == '-':
            L1.append(c)
            L2.append("")
        else:
            L2[-1] += c
    return L1, L2
```

Question 11.b – On vérifie avec la fonction `verifCarac` que chaque élément de `L2` représente soit un entier positif, soit un nom de variable. Si c'est un nom de variable, on vérifie que c'est une clé de `dVar`.

En particulier, si `s` contient une parenthèse, `L2` contient une parenthèse et donc la fonction renvoie `False`. Si `s` est vide ou bien si son premier caractère est `"-"` alors `L2[0]` est la chaîne de caractères vide et donc la fonction renvoie `False`.

```
def estExprNorm(s, dVar):
    """estExprNorm(s: str, dVar: dict[str, int]) -> bool"""
    _, L2 = decomposer(s)
    for s in L2:
        if not (verifCarac(s, C_chiffres) or verifCarac(s, C_lettres)):
            return False
        if verifCarac(s, C_lettres) and not s in dVar:
            return False
    return True
```

Question 11.c –

```
# Suppose que s représente un entier positif ou un nom de variable
def evalOperande(s, dVar):
    """evalOperande(s: str, dVar: dict[str, int]) -> int"""
    if verifCarac(s, C_chiffres):
        return strToInt(s)
    else:
        return dVar[s]
```

```
def evalExprNorm(s, dVar):
    """evalExprNorm(s: str) -> int"""
    assert estExprNorm(s, dVar)
    L1, L2 = decomposer(s)
    res = evalOperande(L2[0], dVar)
    for i in range(len(L1)):
        x = evalOperande(L2[i+1], dVar)
        if L1[i] == '+':
            res = res + x
        else:
            res = res - x
    return res
```

Question 12 –

```
def evalExpr(s, dVar):
    """evalExpr(s, dVar) -> int"""
    assert bienParenthesee(s)
    s = normaliser(s)
    return evalExprNorm(s, dVar)
```

Remarque. Si `s` n'est pas une expression arithmétique, le comportement de `evalExpr` peut être inattendu. Par exemple, pour `s = "(1)1"`, on obtient 11.

Question 13 –

```
# Suppose que s représente une affectation
def affectation(s, dVar):
    """affectation(s: str, dVar: dict[str, int]) -> NoneType"""
    d = strToDict(s)
    L = d['=']
    i = L[0]
    nom_var = s[:i]
    expr = s[i+1:]
    valeur = evalExpr(expr, dVar)
    dVar[nom_var] = valeur
```

Question 14 –

```
def executer(prog):
    """executer(prog: str) -> NoneType"""
    dVar = {}
    prog = supprSPC(prog)
    L = separer(prog)
    L = [supprCom(s) for s in L]
    L = supprVides(L)
    for s in L:
        dict_s = strToDict(s)
        if "=" in dict_s:
            affectation(s, dVar)
        else:
            valeur = evalExpr(s, dVar)
            print(valeur)
```