Séance 7 : Modules et types abstrait

L1 – Université Côte d'Azur

Exercice 1 — Un générateur pseudo-aléatoire (⋆)

On considère la suite à valeurs dans $\{0, 1, \dots, 9\}$, définie par la récurrence $u_{n+1} = 2u_n + 1$ modulo 11, et par $u_0 = 3$.

1. À l'aide d'une variable globale, définissez une fonction que vous nommerez `randint()` et qui renvoie u_n la $n$ième fois qu'on l'appelle. Par exemple `[randint(), randint(), randint()]` renverra `[2, 5, 6]` (en supposant que `randint` n'a pas encore été appelée auparavant).
2. On suppose que cette fonction a été écrite dans un fichier qui s'appelle `pseudo_random.py`.
   (a) Comment feriez-vous au toplevel pour appeler cette fonction `randint` ?
   (b) pour appeler la fonction `randint` du module `random` ?
   (c) pour appeler les deux dans une même instruction en laissant à chacune son nom d'origine ?
   (d) en renommant notre fonction `randint` en `pseudo_randint` ?

Exercice 2 — Le type abstrait file (⋆⋆)

Une file est un conteneur de valeurs qui fonctionne en mode FIFO (*first in, first out*, i.e. premier ajouté premier sorti). Les opérations élémentaires sur une file sont (1) créer une nouvelle file vide, (2) tester si une file est vide, (3) ajouter un élément en fin de file, (4) retirer (et renvoyer) l'élément en début de file.

Définissez un module `file.py` qui donne une représentation en Python de ce type abstrait en utilisant une liste Python pour représenter une file. On rappelle que `L.insert(i, v)` insère `v` à l'indice `i` de `L` et que `L.pop(i)` retire l'élément d'indice `i` de la liste `L`.

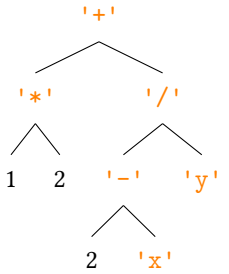
Important : il y a deux approches possibles, vous en choisirez une, et surtout vous veillerez à ce que quelqu'un qui veut utiliser votre module n'ait pas besoin de savoir quel choix vous avez fait.

Dans les exercices suivants, on manipule des arbres binaires d'expression (`abe`) définis dans le cours : un arbre est soit une feuille (nombre ou variable), soit un nœud avec une racine opérateur et deux fils. On ne s'intéresse pas à la façon dont les arbres sont représentés, on va utiliser uniquement les fonctions suivantes, disponibles dans le module `abe.py` (cours 7 p. 37).

- `arbre(r,Ag,Ad)` qui renvoie un arbre de racine `r` et de fils `Ag` (gauche) et `Ad` (droit) ;
- `est_feuille(op)` qui renvoie `True` si `op` est une feuille, et `False` sinon ;
- `racine(A)` qui renvoie la racine de l'arbre `A`
- `fg(A)` et `fd(A)` qui renvoient respectivement le fils gauche et le fils droit de l'arbre `A`.

Exercice 3 — Arbuste (⋆)

On considère l'arbre ci-contre.



1. À quelle expression mathématique correspond-il ?
2. Quelles sont ses feuilles ?
3. Combien a-t-il de nœuds ?
4. Est-ce un arbre binaire complet ?
5. Comment le définir avec les fonctions du module `abe.py` ?
6. Quel est son parcours profondeur préfixe ?

Exercice 4 — Nombre d'opérateurs d'un arbre (⋆⋆)

Définissez par récurrence la fonction `nombre_opérateurs(A)` qui renvoie le nombre d'opérateurs de `A` (c'est-à-dire le nombre de nœuds).

```
def nombre_opérateurs(A):
    if est_feuille(A):
        return 0
    else:
        g = nombre_opérateurs(fg(A))
        d = nombre_opérateurs(fd(A))
        return 1 + g + d
```

Exercice 5 — Arbres arithmétiques (⋆⋆)

Définissez par récurrence la fonction `est_arithmétique(A)` qui renvoie `True` si l'arbre `A` est arithmétique — autrement dit s'il ne contient aucune variable — et `False` sinon.

```
def est_arithmétique(A):
    if est_feuille(A):
        return type(A) == int
    else:
        g = est_arithmétique(fg(A))
        d = est_arithmétique(fd(A))
        return g and d
```

Si `g` est `False`, il est inutile de faire de le calcul de `d`. Pour éviter de faire un calcul inutile, on fait les appels récursifs sur le ligne du `and`. Ainsi si le premier appel récursif renvoie `False`, le second appel récursif ne sera pas fait.

```
def est_arithmétique(A):
    if est_feuille(A):
        return type(A) == int
    else:
        return est_arithmétique(fg(A)) and est_arithmétique(fd(A))
```

Exercice 6 — Plus petite feuille d'un arbre (⋆⋆– ⋆ ⋆ ⋆)

Définissez par récurrence la fonction `min_feuille(A)` qui renvoie la valeur du plus petit entier apparaissant sur une feuille de l'arbre :

1. d'abord en supposant que l'arbre `A` est arithmétique ;

```python
def min_feuille(A):  # A est supposé arithmétique
    if est_feuille(A):
        return A
    else:
        mg = min_feuille(fg(A))
        md = min_feuille(fd(A))
        return min(mg, md)
```

2. ensuite pour un arbre quelconque ; vous traiterez par une exception le cas où l'arbre ne contient que des variables sur les feuilles.

```python
def min_feuille2(A): # A n'est plus supposé arithmétique
    if est_feuille(A):
        if type(A) == int:
            return A
        else:
            raise ValueError('Pas de feuille entière')
    else:
        try:
            mg = min_feuille2(fg(A))
        except:
            return min_feuille2(fd(A))
        try:
            md = min_feuille2(fd(A))
        except:
            return mg
        return min(mg, md)
```

Autre solution : on utilise une fonction auxiliaire qui renvoie - infini lorsque le min n'est pas defini, puis on leve l'exception dans la fonction englobante

```python
import math

def min_feuille2_2(A) :

    def aux(A) :
        if est_feuille(A) :
            if type(A) == int :
                return A
            else :
                return math.inf  # infini
        else :
            return min(aux(fg(A)), aux(fd(A)))

    res = aux(A)
    if res == math.inf :
        raise ValueError("pas de feuille entiere")
    else :
        return res
```

Exercice 7 — Parcours infixe d'un arbre (★★– ★ ★ ★)

Le parcours profondeur infixe d'un arbre binaire d'expression est un parcours en profondeur dans lequel on visite d'abord le fils gauche, puis la racine, puis le fils droit. Par exemple, pour l'arbre qui correspond à l'expression « `(1*2)-(3/4)` », on aura le parcours profondeur infixe `[1, '*', 2, '-', 3, '/', 4]`. Autrement dit, le parcours profondeur infixe correspond à la suite de symboles dans l'écriture mathématique habituelle, sans les parenthèses.

1. Définissez une fonction `pp_infixe(A)` qui renvoie le parcours infixe de `A`.

```python
def pp_infix(A):
    if est_feuille(A):
        return [A]
    else:
        return pp_infix(fg(A)) + [racine(A)] + pp_infix(fd(A))
```

2. En vous inspirant de `pp_infixe(A)`, définissez la fonction `repr_math(A)` qui renvoie la représentation mathématique de `A` sous forme d'une chaîne de caractères, en ajoutant les parenthèses pour chaque sous-arbre strict qui n'est pas une feuille. Par exemple `repr_math(A)` renverra `'(1*2)-(3/4)'` pour l'arbre de l'exemple ci-dessus.

   Version naïve qui rajoute des parenthèses autour de l'expression finale.

```python
def repr_math_0(A):
    if est_feuille(A):
        return str(A)
    else:
        rg = repr_math(fg(A))
        rd = repr_math(fd(A))
        o  = racine(A)
        return f'({rg}{o}{rd})'
```

   On rajoute des parenthèses sauf si rg (ou rd) est une feuille

```python
def repr_math(A):
    if est_feuille(A):
        return str(A)
    else:
        if est_feuille(fg(A)):
            rg = repr_math(fg(A))
        else:
            rg = '(' + repr_math(fg(A)) + ')'

        if est_feuille(fd(A)):
            rd = repr_math(fd(A))
        else:
            rd = '(' + repr_math(fd(A)) + ')'

        o  = racine(A)
        return f'{rg}{o}{rd}'
```

Exercice 8 — Le parcours en profondeur préfixe est réversible (⋆ ⋆ ⋆)

Dans le cours, on a vu une fonction `pp_prefixe(A)` qui permet d'« aplatir » l'arbre arithmétique `A`, c'est-à-dire de produire la liste des nœuds et feuilles de `A` par un parcours en profondeur préfixe de `A`.

Écrivez une fonction réciproque `arboriser(L)` qui à partir de la liste des nœuds renvoie un arbre `A` tel qu'on ait `pp_prefixe(A) == L`.

Indication : On part d'une liste `L` qui contient des arbres (les feuilles) et des opérations. À chaque fois que l'on rencontre dans la liste, une opération suivie de deux arbres (qui peuvent être des feuilles), on remplace ces trois éléments par un seul arbre. Exemple :

```
L = [ ... , '+', A1, A2, ... ]
L = [ ... , arbre('+', A1, A2), ... ]
```

On recommence jusqu'à ce que la liste `L` soit réduite à un seul élément.

```python
def arboriser(A): # A est une liste
    while len(A)>2:
        L=[]
        i=0
        while i<len(A)-2:
            if type(A[i]) == str and type(A[i+1]) !=str  and type(A[i+2]) != str :
                L.append(arbre(A[i],A[i+1],A[i+2]))
                i=i+2
            else:
                L.append(A[i])
            i=i+1
            print(L)
        A=L
    return A.pop()
```