Examen
Corrigé

NOM : **Prénom :**

Consignes : Téléphones, ordinateurs et calculatrices interdits.
Documents de cours/TD/TP autorisés.
Aucune question possible : en cas d'ambiguïté dans le sujet, signaler le choix fait sur la copie.

Conseil : Lire complètement le sujet avant de commencer à y répondre.

Barème indicatif :

exercice	1	2	3	4
points	6	10	2	2

Exercice 1 : Calculer une valeur sur les éléments d'une liste *1*
Exercice 2 : Les expressions arithmétiques revisitées *3*
Exercice 3 . . . . *7*
Exercice 4 . . . . *7*

Exercice 1 : Calculer une valeur sur les éléments d'une liste

Dans cet exercice, on utilisera le langage Python.

1. *Somme.* Écrire une fonction `somme` qui retourne la somme des éléments d'une liste. On n'utilisera pas les fonctions prédéfinies de Python comme par exemple **`sum`**.

Par exemple, appliquée sur la liste [1, -1, 2, 3], cette fonction retournera 5.

Solution :

```
def somme(liste):
    resultat = 0
    for x in liste:
        resultat = resultat + x
    return resultat

def test_somme():
    assert somme([1, -1, 2, 3]) == 5
```

2. *Produit.* Écrire une fonction `produit` qui retourne le produit des éléments d'une liste. On n'utilisera pas les fonctions prédéfinies de Python comme par exemple `Math.prod`.

Par exemple, appliquée sur la liste [1, -1, 2, 3], cette fonction retournera -6.

Solution :

```
def produit(liste):
    resultat = 1
    for x in liste:
        resultat = resultat * x
```

```
5       return resultat
6
7   def test_produit():
8       assert produit([1, -1, 2, 3]) == -6
```

3. *Généraliser.* Les deux fonctions précédentes sont sur le même principe : leur code a la même structure. Proposer une fonction que l'on peut appeler `reduce` qui généralise ces deux fonctions.
Indication : Il s'agit de transformer en paramètres de cette fonction les éléments différents dans les codes des deux fonctions `somme` et `produit`.
Solution :

```
1   def reduce(liste, combiner, valeur_initiale):
2       resultat = valeur_initiale
3       for x in liste:
4           resultat = combiner(resultat, x)
5       return resultat
```

4. *Utiliser `reduce`.*
4.1. Réécrire `somme` en utilisant `reduce`.
Solution :

```
1   def somme_reduce(liste):
2       def _somme(s, x):
3           return s + x
4       return reduce(liste, _somme, 0)
5
6   def produit_reduce(liste):
7       def _produit(p, x):
8           return p * x
9       return reduce(liste, _produit, 1)
10
11  def test_reduce():
12      assert somme_reduce([1, -1, 2, 3]) == 5
13      assert produit_reduce([1, -1, 2, 3]) == -6
```

4.2. En utilisant `reduce`, définir une fonction `frequence` qui calcule la fréquence (le nombre d'occurrences) d'un élément dans une liste. La fréquence de 3 dans [3, 1, 2, 3] est 2.
Solution :

```
1   def frequence(liste, element):
2       def combiner(f, x):
3           if x == element:
4               return f + 1
5           else:
6               return f
7       return reduce(liste, combiner, 0)
8
```

```
def test_frequence():
    assert frequence([1, 2, 3, 1, 2, 1], 1) == 3
    assert frequence([1, 2, 3, 1, 2, 1], 2) == 2
    assert frequence([1, 2, 3, 1, 2, 1], 3) == 1
    assert frequence([1, 2, 3, 1, 2, 1], 4) == 0
```

Exercice 2 : Les expressions arithmétiques revisitées

Dans cet exercice nous modélisons en OCaml, Python objet et Prolog des expressions arithmétiques simplifiées, limitées à la somme et à la différence. Nous proposons la modélisation suivante, en OCaml, des expressions arithmétiques.

```
type operateur =
    | Somme
    | Difference
;;

type expression =
    Constante of float
    | ExpressionBinaire of (operateur * expression * expression)
;;
```

1. Définir en OCaml l'expression $5 - (3 + 4)$ que l'on appellera e1.

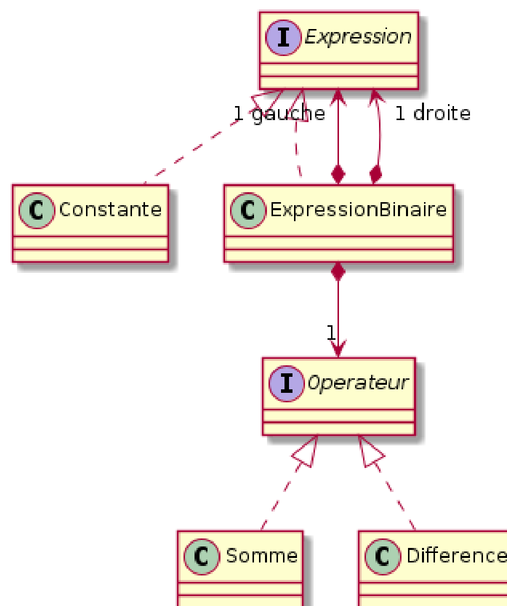
Solution :

```
let e1 = ExpressionBinaire(Difference, Constante 5.0,
        ExpressionBinaire(Somme, Constante 3.0, Constante 4.0));;
```

2. Modélisons les expressions en objet en gardant la même logique.

2.1. Dessiner le diagramme de classe qui correspond à ces expressions artihmétiques.

Solution :



2.2. Écrire en Python l'initialisation de la constante e1 ci-dessus.
Solution :

```
def e1():
    return ExpressionBinaire(Difference(), Constante(5),
                ExpressionBinaire(Somme(), Constante(3), Constante(4)))
```

3. Définir un prédicat Prolog d'arité 1 appelé e1 qui permet d'unifier sa variable avec l'expression e1 précédente.
Solution :

```
e1(X) :- X = expressionBinaire(difference, constante(5),
        expressionBinaire(somme, constante(3), constante(4))).
```

4. *Nombres tirets.* On souhaite faire un premier traitement sur ces expressions qui consiste à obtenir le nombre de tirets utilisés dans les opérateurs d'une expressions.

Ce traitement peut être écrit en OCaml par la fonction `nombre_traits` suivante.

```
let rec nombre_traits e =
    let rec nombre_traits_operateur op gauche droite
        match op with
        | Somme -> 2
        | Difference -> 1
    in
        match e with
        | ExpressionBinaire (op, e1, e2) -> (nombre_traits_operateur op)
                + (nombre_traits e1) + (nombre_traits e2)
        | _ -> 0
;;
```

4.1. Expliquer le code ci-dessus.
Solution : La fonction `nb_traits` prend en paramètre une expression et retourne un entier. C'est l'analyse du code qui permet de le savoir.

Cette fonction définit une fonction locale qui retourne le nombre de traits utilisé pour tracer un opérateur (2 pour la Somme (+) et 1 pour la différence (-)). Le `match` traite ainsi tous les cas possible d'opérateur.

Elle contient une définition de fonction qui prend en paramètre un opérateur et retourne son nombre de traits

Le `match` de la fonction `nb_traits` traite le cas d'une expression binaire : le nombre de traits est alors la somme du nombre de traits de l'expression gauche, de l'expression droite et de l'opérateur.

4.2. Indiquer le résultat de l'application de ce traitement sur l'expression e1.
Solution : 3

4.3. *En objet.* Expliquer comment écrire ce traitement dans la version objet et donner quelques exemples de code significatifs.

Solution : On définit une méthode `nb_traits` sur toutes les classes de nos expressions (Expression et ses sous-classes, Opérateur et ses sous-classes).

Ci-dessous, tout le code correspondant est donné. Il suffirait de donner celui de `ExpressionBinaire` et `Somme`.

```
class Expression(metaclass=abc.ABCMeta):
    @property
    @abc.abstractmethod
    def nb_traits(self):
        ...


class Constante(Expression):

    def __init__(self, valeur):
        self.__valeur = float(valeur)

    @property
    def nb_traits(self):
        return 0


class ExpressionBinaire(Expression):

    def __init__(self, operateur, operande_gauche, operande_droite):
        self.__operateur = operateur
        self.__operande_gauche = operande_gauche
        self.__operande_droite = operande_droite

    @property
    def nb_traits(self):
        return self.__operateur.nb_traits   \
               + self.__operande_gauche.nb_traits \
               + self.__operande_droite.nb_traits


class Operateur(metaclass=abc.ABCMeta):

    @property
    @abc.abstractmethod
    def nb_traits(self):
        ...


class Somme(Operateur):

```

```
42      @property
43      def nb_traits(self):
44          return 2
45
46
47  class Difference(Operateur):
48
49      @property
50      def nb_traits(self):
51          return 1
```

4.4. *En Prolog.* Écrire en Prolog ce traitement.
Solution :

```
1  nb_traits(constante(N), 0).
2  nb_traits(expressionBinaire(O, G, D), N) :-
3          nb_traits(G, NG), nb_traits(D, ND), nb_traits(O, NO), V is NG + ND + NO.
4  nb_traits(somme, 2).
5  nb_traits(difference, 1).
```

5. *Évaluation d'une expression.* On s'intéresse à un nouveau traitement qui consiste à obtenir la valeur d'une expression.
5.1. *En fonctionnel.* Écrire ce traitement en OCaml.
Solution :

```
1  let rec valeur e =
2      let rec appliquer_operateur op gauche droite
3          match op with
4          | Somme -> gauche +. droite
5          | Difference -> gauche -. droite
6          | Produit -> gauche *. droite
7      in
8          match e with
9          | Constante v -> v
10         | ExpressionBinaire (op, e1, e2) ->
11                 (appliquer_operateur op (valeur e1) +. (valeur e2))
12 ;;
```

5.2. Écrire ce traitement en Python. On indiquera le code à ajouter et où l'ajouter.
Solution :

```
1  class Expression(metaclass=abc.ABCMeta):
2      @property
3      @abc.abstractmethod
4      def valeur(self):
5          ...
6  class Constante(Expression):
```

```python
    @property
    def valeur(self):
        return self.__valeur


class ExpressionBinaire(Expression):
    @property
    def valeur(self):
        gauche = self.__operande_gauche.valeur
        droite = self.__operande_droite.valeur
        return self.__operateur.appliquer(gauche, droite)


class Operateur(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def appliquer(self, valeur_gauche, valeur_droite):
        ...


class Somme(Operateur):
    def appliquer(self, valeur_gauche, valeur_droite):
        return valeur_gauche + valeur_droite


class Difference(Operateur):
    def appliquer(self, valeur_gauche, valeur_droite):
        return valeur_gauche - valeur_droite
```

5.3. *En déclaratif.* Écrire ce traitement en Prolog.

Solution :

```prolog
valeur(constante(N), N).
valeur(expressionBinaire(somme, G, D), V) :-
        valeur(G, VG), valeur(D, VD), V is VG + VD.
valeur(expressionBinaire(difference, G, D), V) :-
        valeur(G, VG), valeur(D, VD), V is VG - VD.
```

Exercice 3 Écrire en Prolog un prédicat max qui associe à une liste son plus grand élément.

Solution :

```prolog
max([X], X).
max([X|S], X) :- max(S, MS), MS =< X, !.
max([X|S], R) :- max(S, R), R > X.
```

Exercice 4 Indiquer les solutions qui seront trouvées par Prolog pour la requête `?- b(X).` sur les règles suivantes. Expliquer comment Prolog procède.

```
a(1).
a(3).
a(3).
a(2).
a(6).
a(5).

b(X) :- a(X), X > 2, X < 6.
```

Solution :

Le résultat de l'exécution est :

```
?- b(X).
X = 3 ;
X = 3 ;
X = 5.
```

Le principe : on unifie le but, ici (b(X)) avec la tête des règles dans l'ordre de haut en bas. Quand une règle correspond, on vérifie alors la partie droite qui deviennent de nouveaux buts à vérifier. En cas d'échec, on continue avec les règles suivantes (backtracking).