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
points	4	6	10

Exercice 1 On considère le « programme » suivant :

```
1 v(1).
2 v(3).
3 v(2).
4 f(A, B) :- v(A), v(B), A @< B.
```

1. Que représentent v, f, 1, 2, 3, A, B ?
Solution :
- v et f sont des prédicats qui servent à construire des éléments composés.
- 1, 2 et 3 sont des entiers (élément atomique).
- A et B sont des variables (ou inconnues).

2. Indiquer ce que répondra Prolog à la question suivante :

```
1 ?- f(X, Y).
```

Solution :

```
1 ?- f(A, B).
2 A = 1,
3 B = 3 ;
4 A = 1,
5 B = 2 ;
```

```
A = 2,
B = 3 ;
false.
```

3. Est-ce que la question suivante est possible ? La réponse doit être justifiée soit en expliquant pourquoi ce n'est pas possible, soit en donnant le résultat de l'évaluation.

```
?- f(C, 2).
```

Solution : Cette question est valide.
La réponse sera :

```
C = 1 ;
false.
```

Exercice 2 On veut savoir si une liste contient deux éléments consécutifs égaux. On appellera dce la fonction ou le prédicat qui répond à cette question.

Ainsi, dce appliqué sur [1, 4, 4, 2] est vrai et appliqué sur [3, 1, 3, 4] est faux.

1. Pour vérifier la compréhension de dce, indiquer sa valeur dans les cas suivants :

```
[]
[2, 3]
[2, 2]
[1, 2, 3, 4, 4]
```

Solution : Les réponses sont respectivement : faux, faux, vrai (les deux premiers sont égaux), vrai (les deux derniers sont égaux).

2. Écrire une version itérative (en Python) de `dce`.

Solution :

```python
def dce_iteratif(liste):
    for i in range(len(liste) - 1):
        if liste[i] == liste[i + 1]:
            return True
    else:
        return False
```

3. Écrire une version récursive (en Ocaml ou Python) de `dce`.

Solution : En Ocaml :

```
let rec dce(liste) =
    match (liste) with
    | x :: y :: suite ->
            if x = y then
                true
            else
```

```
                dce (y :: suite)
    | _ -> false
;;
```

En Python :

```
def dce(liste):
    if len(liste) < 2:
        return False
    elif liste[0] == liste[1]:
        return True
    else:
        return dce(liste[1:])
```

4. Écrire une version déclarative (en Prolog) de `dce`.
Solution :

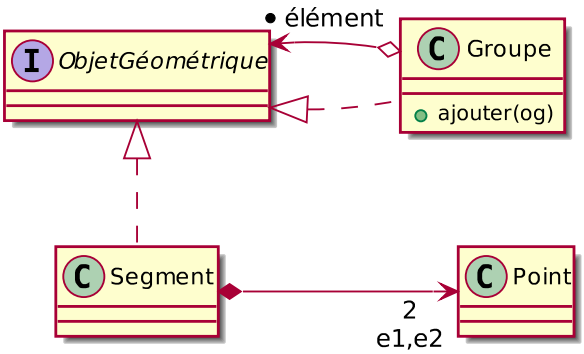
```
dce([X, X | _]).
dce([X | S]) :- dce(S).
```

Exercice 3 On s'intéresse à des objets géométriques qui peuvent être soit un segment (défini par deux points extrémité e1 et e2), soit un groupe composé d'une liste d'objets géométriques. Un point est défini par une abscisse x et une ordonnée y. Un point n'est pas un objet géométrique.

Ainsi, on pourrait définir un premier groupe appelé triangle qui serait constitué des segments (p1, p2), (p2, p3) et (p3, p1) avec p1, p2 et p3 trois points. On pourrait définir un deuxième groupe appelé schéma composé du groupe triangle précédent et d'un segment ayant pour extrémités le point origine (0, 0) et le point(10, 0).

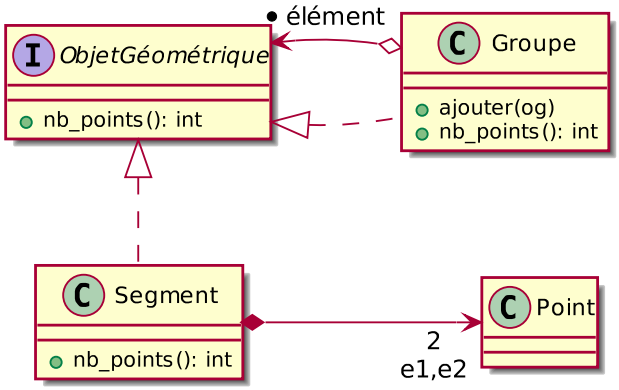
On veut obtenir pour un objet géométrique le nombre de points qu'il utilise sachant qu'un segment en utilise deux points (ses deux extrémités) et un groupe en utilise la somme des points utilisés par les objets géométriques qu'il regroupe. Le nombre de point du triangle précédent est donc 6 (2 + 2 + 2) et celui du schéma est 8 (6 + 2).

1. *Objet.* Commençons par proposer une modélisation objet en UML et Python de ce problème.
1.1. Proposer un diagramme de classe de UML qui représente les objets géométriques.
Solution :



1.2. Compléter le diagramme de classe pour expliquer comment le nombre de points d'un objet géométrique peut être implanté.

Solution :



1.3. Écrire le code Python des classes du diagramme de classe précédent.

Solution :

```python
import abc

class ObjetGeometrique(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def nb_points(self):
        pass

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Segment(ObjetGeometrique):
    def __init__(self, e1, e2):
        self.e1 = e1
        self.e2 = e2

    def nb_points(self):
        return 2

class Groupe(ObjetGeometrique):
    def __init__(self):
        self.elements = []

    def ajouter(self, og):
        self.elements.append(og)

```

```python
    def nb_points(self):
        return sum(o.nb_points() for o in self.elements)

def triangle():
    p1 = Point(3, 2)
    p2 = Point(6, 9)
    p3 = Point(11, 1)
    g = Groupe()
    g.ajouter(Segment(p1, p2))
    g.ajouter(Segment(p2, p3))
    g.ajouter(Segment(p3, p1))
    return g

if __name__ == '__main__':
    print('nb points pour le triangle :', triangle().nb_points())
```

2. *Fonctionnel.* Proposons maintenant une modélisation fonctionnelle sachant que l'on a décidé de définir comme suit les types :

```
type point =
    Point of (int * int)
;;

type objet_geometrique =
    | Segment of (point * point)
    | Groupe of objet_geometrique list
;;
```

2.1. Définir les points p1, p2 et p3 de coordonnées respectives (3, 2), (6, 9), (11, 1) puis le triangle.
Solution :

```
let p1 = Point(3, 2) and p2 = Point(6, 9) and p3 = Point(11, 1);;

let triangle = Groupe([Segment(p1, p2) ; Segment(p2, p3) ; Segment(p3, p1)]);;
```

2.2. Écrire la fonction qui donne le nombre de points utilisés par un objet géométrique.
Solution :

```
let rec nb_points(og) =
    let rec nb_pts (liste) =
        match (liste) with
        | [] -> 0
        | o1 :: suite -> (nb_points o1) + (nb_pts suite)
    in
```

```
7           match (og) with
8           | Segment(_, _) -> 2
9           | Groupe(objets) -> nb_pts objets
10  ;;
```

3. *Logique.* Proposons aussi une modélisation en programmation logique.
3.1. Expliquer comment représenter l'objet géométrique triangle décrit ci-avant.
Solution : On peut définir un prédicat pour représenter un point : point/2, un pour un segment : segment/2 et un pour un groupe : groupe/1.

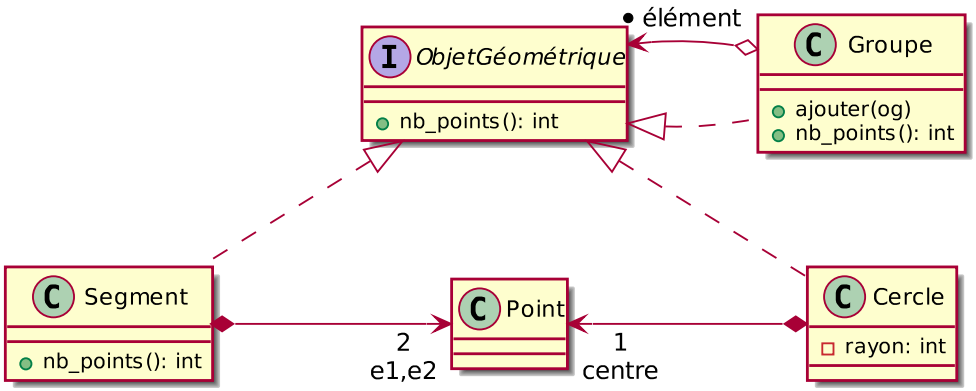
```
1  p1(X) :- X = point(3, 2).
2  p2(X) :- X = point(6, 9).
3  p3(X) :- X = point(11, 1).
4
5  triangle(X) :-
6          p1(P1), p2(P2), p3(P3),
7          X = groupe([segment(P1, P2), segment(P2, P3), segment(P3, P1)]).
```

3.2. Écrire un prédicat qui indique le nombre de points utilisés par un objet géométrique.
Solution :

```
1  % nb_points(X, N) : N est le nombre de points utilisés dans X
2  nb_points(segment(A, B), 2).
3  nb_points(groupe([]), 0).
4  nb_points(groupe([X | S]), N) :-
5          nb_points(X, NX), nb_points(groupe(S), NS), N is NX + NS.
```

4. *Ajout d'un objet géométrique.* Indiquer pour les trois paradigmes (objet, fonctionnel et logique) ce qu'il faudrait faire pour ajouter un cercle, nouvel objet géométrique défini à partir de son centre (un point) et de son rayon (un entier).
Solution :

— Pour la partie objet : On ajoute une nouvelle classe, Cercle, qui hérite de ObjetGéométrique et définit la méthode `nb_points` pour renvoyer



1.

— Pour la partie fonctionnel, il faut rajouter un nouveau cas d'objet géometrique

```
| Cercle of (point * int)
```

et un nouveau cas dans le match de `nb_points` pour renvoyer 1 dans le cas d'un cercle.

— Pour la partie déclarative, on ajoute un nouveau prédicat cercle/2 qui prend un point et un entier. On rajoute une règle :

```
nb_points(Cercle(C, R), 1).
```