Nom : TRAORE
Prénom : Oumar
IDBooster : 165366
Projet : 3AIT

Exercice 1

Prérequis :
IDE : Lispwork version 6.1.1
OS : OSX Yosemite

Question 1.1 :
Ecrire une fonction (reorganiser1 '(les mouton) '(le loups)) -> '((les loups) le mouton)

```
(defun reorganise1 (liste1 liste2)
        (setq a  (car liste1))
        (setq b  (cadr liste1))
        (setq c  (car liste2))
        (setq d  (cadr liste2))
        (list (list a d) c b)
)
```

Question 1.2 :
Ecrire une fonction (reorganise2 '(les mouton) '(le loups)) -> '(les loups le mouton)

```
(defun reorganise2 (liste1 liste2)
        (setq a  (car liste1))
        (setq b  (cadr liste1))
        (setq c  (car liste2))
        (setq d  (cadr liste2))
        (list a d c b)
)
```

Question 1.3 :
Ecrire une fonction (reorganise3 '(les mouton) '(le loups)) -> '((les loups) (le mouton))

```
(defun reorganise3 (liste1 liste2)
        (setq a  (car liste1))
        (setq b  (cadr liste1))
        (setq c  (car liste2))
        (setq d  (cadr liste2))
        (list (list a d) (list c b))
)
```

Question 1.4 :
Ecrire une fonction (dupliquer 'a) -> (a a a)

```
(defun duplique (atome)
```

```
        (list atome atome atome)
)
```

Question 1.5 :

Ecrire une fonction (construireListe '(a) 'a '(a)) -> ((a) a (a))

```
(defun construireListe (atome1 atome2 atome3)
        (list atome1 atome2 atome3)
)
```

Question 1.6 :

Que donnent les interprétations suivantes:

1- (reorganise1 (construireListe 'UN 'LA 'PHRASE) (reorganise1 (construireListe 'AVEZ 'BRAVO 'VOUS)(duplique 'GRAND)))

Réponse : ((UN GRAND) (AVEZ GRAND) LA)

2 - (reorganise2 (construireListe 'UN 'LA 'PHRASE) (reorganise2 (construireListe 'AVEZ 'BRAVO 'VOUS)(duplique 'GRAND)))

Réponse: (UN GRAND AVEZ LA)

3- (reorganise3 (construireListe 'UN 'LA 'PHRASE) (reorganise3 (construireListe 'AVEZ 'BRAVO 'VOUS)(duplique 'GRAND)))

Réponse: ((UN (GRAND BRAVO)) ((AVEZ GRAND) LA))

Question 1.7 :

Quelle interprétation, selon vous, à l'aide de
La Question 1.6 permet d'avoir le meilleur résultat, expliquez et détaillez.

Analysons le retour de :
(duplique 'GRAND) -> (GRAND GRAND GRAND)
(construireListe 'AVEZ 'BRAVO 'VOUS) -> (AVEZ BRAVO VOUS)

Exercice 2

Question 1.2

;;
Spécification :

* *Abscisse* représenté par L'Atome : x
* *Ordonnée* représenté par L'Atome : y

* *Point* est une liste composée des Atomes Abscisse et Ordonnée : (x y)

* *Segment* est une liste composé de 2 Points : (Point1 Point2) -> ((x1 y1) (x2 y2))
Avec :
  Point1 : (x1 y2)
  Point2 : (x2 y2)

* PointMilieu est représenté par un point : PointMilieu = (Point) -> (x3 y3)
  Les Atomes du PointMilieu x3 et y3 sont calculer de manière mathématique à l'aide des points (Point1 Point2) du segment qui le compose.
Avec :
    x3 : (x1 +x2)/2
    y3 : (y1 +y2)/2
  PointMilieu est donnée par la formule Mathématique: ((x1 + x2)/2 (y1 + y2)/2))
  En Langage Lisp : ((/ (+ x1 x2) 2) ((/ (+ y1 y2) 2))
;;

;;
Le problème posé :

Ecrire une fonction pointMilieu qui reçoit un segment en paramètre et renvoi en sorti le point Milieu

La Fonction :
pointMilieu(segment) -> PointMiLieu
pointMilieu(segment) -> (x3 y3).

En Langage Lisp

```
(defun pointMilieu(segment)
        (list x 3 y3)
)
```
;;

;;

Processus de calcul des atomes x3 et y3 :

NB : Selon Spécification Ci-dessus.

Nous avons donc :

pointMilieu(segment) -> (x3 y3).
pointMilieu(segment) - > ((x1 + x2)/2 (y1 + y2)/2)

- Cela revient à trouver x1 y1 x2 y2 dans le segment.

Phase 1 : Sous fonctions

Elément participant à la réalisation :

- Fonction Abscisse du point Milieu à partir du segment :
(pMA segment) -> x3, Pour le calcul de x3 = (x1 + x2)/2

- Fonction Ordonnée du Point Milieu à partir du segment:
(pMO segment) - > y3, Pour le calcul de y3 = (y1 + y2)/2

En Langae Lisp nous avons :

```
(defun pointMilieu(segment)
	(list (pMA segment) (pMO segment))
)
```

Sous fonction pMA

```
(defun pMA(segment)
	(/ (+ x1 x2) 2)
)
```

Sous fonction pMO

```
(defun pM0(segment)
	(/ (+ x1 x2) 2)
)
```

- Fonction décompose à droite Nom : D
- Fonction qui décompose à Gauche nommée : G

```
(defun G(element)
	(car element)
)

(defun D(element)
	(cdr element)
)
```

Phase 2 : Détermination x1 y1 x2 y2

Récupérer le Point1 dans le segment : (G segment)
Récupérer Abscisse Point1, x1 : (G (G segment))
Récupérer Ordonnée Point1, y1 : (G (D (G segment)))

Récupérer le Point2 : (G (D segment))
Récupérer Abscisse Point2, x2 : (G (G (D segment)))
Récupérer Ordonnée Point2, y2 : (G (D (G (D segment))))

Réécrire de nos Sous fonctions pMA et pMO avec les valeurs de x1 y1 x2 y2.

```
(defun pMA(segment)
	(/ (+ (G (G segment))  (G (G (D segment)))  ) 2)
)

(defun pMO(segment)
	(/ (+ (G (D (G segment)))  (G (D (G (D segment))))  ) 2)
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Question 1.1

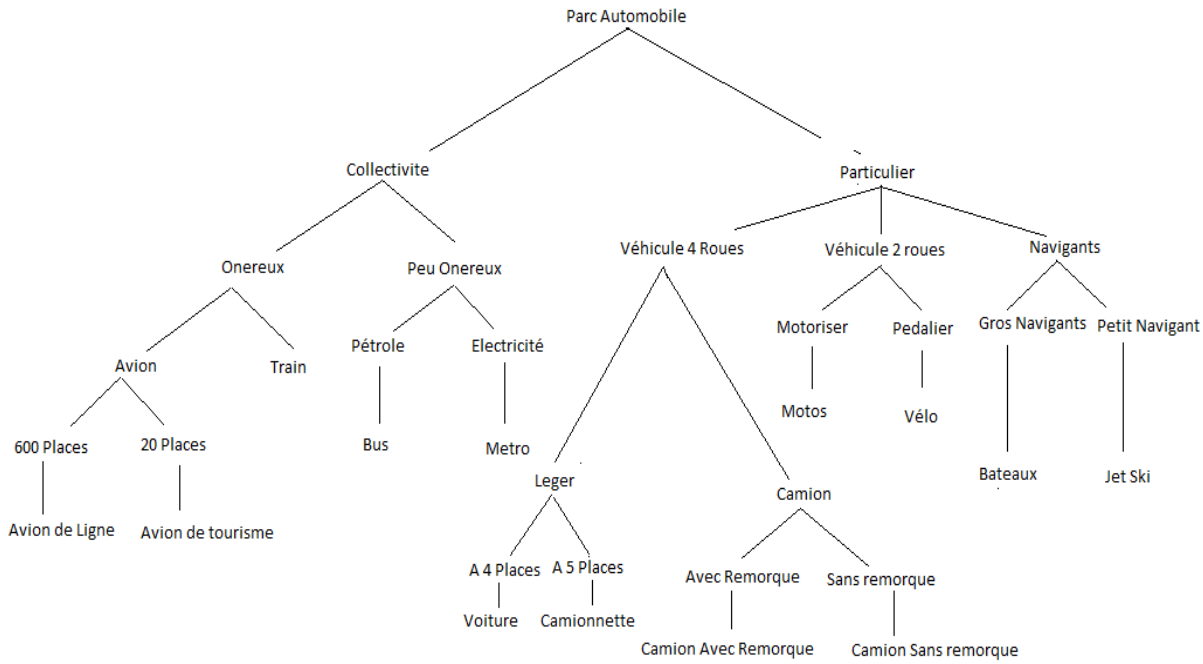
Ecrire le programme Lisp pointMilieur 'segment -> 'Point

```
; Sous Fonction
(defun G(segment)
	(car segment)
)

(defun D(segment)
	(cdr segment)
)

(defun pMA(segment)
	(/ (+ (G (G segment))  (G (G (D segment)))  ) 2)
)

(defun pMO(segment)
	(/ (+ (G (D (G segment)))  (G (D (G (D segment))))  ) 2)
)

; Fonction PointMilieu
(defun pointMilieu(Segment)
	(list (pMA segment) (pMO segment))
)
```

Exercice 3

Question 3.1 :

Moteur d'inférence sous forme d'arbre (Fait avec le Logiciel Paint).



Question 3.2 :

Les règles et diagnostics.

Règles :

Regle1 : Si Collectivité **Et** Onéreux **Alors** Avion
Regle2 : **Si** Avion **Et** 600 Places **Alors** Avion de ligne
Regle3 : **Si** Avion **Et** 20 Places **Alors** Avion de tourisme
Regle4 : **Si** Collectivité **Et** Onéreux **Alors** Train
Regle5 : **Si** Collectivité **Et** Peu Onéreux **Et** Pétrole **Alors** Bus
Regle6 : **Si** Collectivité **Et** Peu Onéreux **Et** Electricité **Alors** Metro
Regle7 : **Si** Particulier **Et** Véhicule a 4 Roues **Alors** Camion
Regle8 : **Si** Camion **Et** Avec Remorque **Alors** Camion Avec Remorque
Regle9 : **Si** Camion **Et** Sans Remorque **Alors** Camion Sans Remorque
Regle10 : **Si** Particulier **Et** Véhicule a 4 Roues **Alors** Leger
Regle11 : **Si** Leger **Et** 4 Places **Alors** Voiture
Regle12 : **Si** Leger **Et** 5 Places Model **Alors** Camionnette
Regle13 : **Si** Particulier **Et** Véhicule a 2 Roues **Et** Pédalier **Alors** Vélo
Regle14 : **Si** Particulier **Et** Véhicule a 2 Roues **Et** Motoriser **Alors** Motos
Regle15 : **Si** Particulier **Et** Navigants **Et** Gros Navigants **Alors** Bateaux
Regle16 : **Si** Particulier **Et** Navigants **Et** Petit Navigants **Alors** Jet Ski

Diagnostics :

Diagnostics1 : Avion de ligne 600 Places
Diagnostics2 : Avion de tourisme 20 Places
Diagnostics3 : Train
Diagnostics4 : Bus
Diagnostics5 : Metro
Diagnostics6 : Voiture
Diagnostics7 : Camionnette
Diagnostics8 : Camionnette
Diagnostics9 : Camionnette sans remorque
Diagnostics10 : Camionnette avec remorque
Diagnostics11 : Moto
Diagnostics12 : Vélo
Diagnostics13 : Bateaux
Diagnostics14 : Jet Ski.

Question 1.3 :

Ecrire le programme Python nécessaire à la réalisation du système expert.

Prérequis :
IDE : IDLE Python : Version 3.4.3
OS : OSX Yosemite

```
#! /usr.bin/env python
# -*-coding:utf8-*-

'''
Une règle est un tuple formé de deux éléments :
- un tuple de string. Les premisses de la règle
- un string. La conclusion de la règle.
Les règles sont rangées dans une liste.
'''
regles = [
    (('Collectivite', 'Onereux',),
        'Avion'),
    (('Collectivite', 'Onereux',),
        'Train'),
    (('Collectivite', 'Peu Onereux', 'Petrole',),
        'Bus'),

    (('Collectivite', 'Peu Onereux', 'Electricite',),
        'Metro'),
    (('Particulier','Vehicule a 4 Roues', 'Leger',),
        'Voiture'),

    (('Particulier','Vehicule a 4 Roues', 'Leger',),
        'Camionnette'),

    (('Particulier','Vehicule a 4 Roues', 'Camion',),
```

```
        'Camion Avec Remorque'),

    (('Particulier','Vehicule a 4 Roues', 'Camion',),
        'Camion Sans Remorque '),

    (('Particulier','Vehicule a 2 Roues', 'Motoriser',),
        'Moto'),

    (('Particulier','Vehicule a 2 Roues', 'Pedalier',),
        'Velo'),

    (('Particulier','Navigants', 'Gros Navigants',),
        'Bateau'),
    (('Particulier','Navigants', 'Petit Navigants',),
        'Jet Ski'),
]

'''
La fonction dansalors permet de trouver les règles accociées à une conclusion
'''

def dansalors(fait):
    results = list()
    for premisses, conclusion in regles:
       if conclusion == fait:
          results.append((premisses, conclusion))
    return results

'''
Initialisation de la mémoire et des faits initiaux
'''

memoire = {}

faits_initiaux = {
'animal est oiseau': True,
}

'''
La fonction connais interroge la mémoire (et les faits initiaux)
'''

def connais(fait):
    resultat = None
    # interrogation des faits prédefinis
    if faits_initiaux: resultat = faits_initiaux.get(fait, None)

    # Interrogation des faits mémorisés
    if resultat == None and memoire: resultat = memoire.get(fait, None)
```

```
    return resultat

'''
La fonction memorise sauvegarde un fait dans la mémoire
'''

def memorise(fait, resultat):
    global memoire
    memoire[fait] = resultat

'''
La fonction demander interroge l'utilisateur.
'''

def demander(fait, question='Est-il vrai que'):
    REPONSES={'o': True, 'n': False,}
    while True:
        choice = input("%s '%s' ?[o/n]" % (question, fait)).lower()
        if choice in REPONSES.keys(): return REPONSES[choice]
        else: print ("Merci de repondre avec o ou n")

'''
La fonction justifie qui, de manière récursive, vas parcourir les règles en
profondeur pour en déduire le but.
'''

def justifie(fait):
    #controle du fait en mémoire
    resultat = connais(fait)
    if resultat != None:
        return resultat

    # détermination des règles possible pour valider le fait courant
    regles = dansalors(fait)

    # Si nous sommes en présence d'une racine, poser la question
    if not regles:
        resultat = demander(fait)
        memorise(fait, resultat)
        return resultat

    # évaluation des règles
    for premisses, conclusion in regles:
        valider = True
        for f in premisses:
            # parcours en profondeur
            if not justifie(f):
                valider = False
```

```
                break
        if valider:
            print (" %s donc %s" % (" et ".join(premisses), fait))
            memorise(fait, True)
            return True

    # aucun fait ou règle trouvé
    return False


'''
La fonction depart qui cherche à prouver un des diagnostics
'''

def depart(diagnostics):
    # parcours depuis les faits diagnostics, les feuilles
    for fait in diagnostics:
        if justifie(fait):
            print ("Conclusion : donc %s" % fait)
            return True

    print ("Aucun diagnostics ne peut etre obtenu")
    return False


'''
Main programme Principale
'''

if __name__ == "__main__":

    # Affichage des règles
    print ("--- Règles chargées:")
    for premisses, conclusion in regles:
        print("Si %s alors %s" % (" et ".join(premisses), conclusion))
    print("---")

    # nous déterminons les feuilles de l'arbre.
    diagnostics = []
    for premisses, conclusion in regles:
        feuille = True
        for p, c in regles:
            if conclusion in p:
                feuille = False
                break
        if feuille:
            diagnostics.append(conclusion)

    # Affichage des diagnostics
```

```
print ("--- Diagnostics")
print (diagnostics)
print ("----")

depart(diagnostics)
```