Corrigé du TD 2

On retiendra la manière « standard » de parcourir une liste chaînée `L`, en utilisant un pointeur `P` qui se déplace « le long de » la liste :

```
P : pointeur vers noeud ; P = L
tant que P<>None faire
    # ici on fait quelque chose avec P->valeur
    P = P->suivant
```

À la sortie de la boucle on a `P = None` et le pointeur `L` pointe toujours vers le début de la liste. On peut comparer la boucle de parcours de liste ci-dessus au parcours d'un tableau `T` rédigé avec une boucle `tant que` comme suit :

```
i : entier ; i = 0
tant que i < taille - 1 faire
    # ici on fait quelque chose avec T[i]
    i = i + 1
```

Exercice 1.

Question 1. *Écrire les instructions nécessaire pour créer une liste vide* `L1` *et la liste* `L2` $= (2 \to 3)$.

```
L1, L2 : liste
L1 = None
L2 = Nouveau(noeud)
L2->valeur = 2
L2->suivant = Nouveau(noeud)
L2->suivant->valeur = 5
L2->suivant->suivant = None
```

Question 2. *Détailler l'exécution du programme* `L : liste ; L = creerListe(3)`*, avec :*

```
creerListe(n : entier) : liste
    nouvelleListe : liste
    nouvelleListe = None
    tmp : pointeur sur noeud
    pour i de n à 1 par pas de -1 faire
        tmp = Nouveau(noeud)
        tmp->valeur = i
        tmp->suivant = nouvelleListe
        nouvelleListe = tmp
    retourner nouvelleListe
```

Au cours de l'exécution de ce programme on a : `nouvelleListe` $= ()$, `nouvelleListe` $= (3)$, `nouvelleListe` $= (2 \to 3)$, `nouvelleListe` $= (1 \to 2 \to 3)$, `L` $= (1 \to 2 \to 3)$.

Question 3. *Dessinez la liste L après la deuxième et la cinquième instruction du programme suivant :*

```
L : liste
L = creerListe(5)
P : pointeur sur noeud
P = L
P->suivant->val = 8
```

On a `L` $= (1 \to 2 \to 3 \to 4 \to 5)$ puis `L` $= (1 \to 8 \to 3 \to 4 \to 5)$.

Question 4. *Que fait la procédure* `mystere` *suivante ? Exprimez en fonction de la longueur de la liste le nombre de fois que l'on effectue l'instruction* `L = L->suivant`*. Donnez une version récursive de* `mystere`*.*

```
mystere(L : liste) : entier
    compteur : entier
    compteur = 0
    tant que L <> None faire
        compteur = compteur + 1
        L = L->suivant
    retourner compteur
```

`mystere` calcule la longueur de `L`. L'instruction `L = L->suivant` est executée autant de fois que la longueur de `L`. Version récursive :

```
longueurRec (L : liste) : entier
    si L == None alors
        retourner 0
    sinon
        retourner 1 + longueurRec(L->suivant)
```

Question 5. *Écrire une procédure itérative qui prend en argument une liste et qui affiche dans le même ordre les éléments de la liste qui sont de valeur paire. On définit comme coût le nombre de nœuds visités. Quel est le coût de la procédure ?*

On utilise l'opérateur `%` qui calcule le reste de la division euclidienne. On a `x % 2 == 0` **ssi** `x` est pair.

```
affichePairsIt (L : liste)
    P : pointeur vers noeud ; P = L
    tant que P <> None faire
        si P->valeur % 2 == 0 alors
            afficher P->valeur
        P = P->suivant
```

Le coût est égal à la longueur de la liste.

Question 6. *Donner une version récursive de la procédure précédente.*

```
affichePairsRec (L : liste)
    si L <> None alors
        si L->valeur % 2 == 0 alors
            afficher L->valeur
        affichePairsRec(L->suivant)
```

Question 7. *Écrire une procédure récursive qui prend en argument une liste et qui affiche dans l'ordre inversé les éléments de la liste qui sont de valeur paire.*

Pour afficher les valeurs paires à l'envers avec une procédure récursive, il suffit de placer l'appel récursif (affichage des valeurs « suivantes ») avant l'affichage de la valeur « en cours ». C'est plus compliqué à réaliser de manière itérative...

```
affichePairsInverseRec (L : liste)
    si L <> None alors
        affichePairsInverseRec(L->suivant)
        si L->valeur % 2 == 0 alors
            afficher L->valeur
```

Question 8. *Écrire une procédure itérative qui prend en argument une liste* `L` *et un entier* `x` *et qui retourne la liste* `L` *obtenue après insertion de* `x` *en fin de la liste. En donner une version récursive.*

Les fonctions suivantes modifient également la liste passée en argument, sauf si elle est vide.

```
ajoutEnFinIt (L : liste, x : entier) : liste
    # on prépare le noeud à rajouter :
    fin : liste ; fin = Nouveau(noeud)
    fin->valeur = x ; fin->suivant = None
    # cas de la liste vide :
    si L == None alors
        retourner fin
    # on parcourt L sans rien faire, en s'arrêtant juste avant la fin :
```

```
    P : pointeur vers noeud ; P = L
    tant que P->suivant <> None faire
        P = P->suivant
    # c'est ici qu'on rajoute le nouveau noeud à la fin de L :
    P->suivant = fin
    retourner L

ajoutEnFinRec (L : liste, x : entier) : liste
    # cas de la liste vide, qui sert quand on est arrivé au bout de L dans la récursion :
    si L == None alors
        fin : liste ; fin = Nouveau(noeud)
        fin->valeur = x ; fin->suivant = None
        retourner fin
    # si la liste n'est pas vide, on applique simplement la même fonction à la suite de la liste :
    L->suivant = ajoutEnFinRec(L->suivant, x)
    retourner L
```

Question 9. *Écrire une procédure qui retourne* `Vrai` *si les valeurs des nœuds de la liste sont rangés par ordre croissant et* `Faux` *sinon. Par convention, on retourne* `Vrai` *si la liste est vide ou constituée d'un seul nœud.*

Voici une solution récursive. Noter l'utilisation de `et` comme opérateur booléen.

```
testCroissant (L : liste) : booléen
    si L == None ou L->suivant == None alors
        retourner Vrai
    retourner (L->valeur <= L->suivant->valeur) et testCroissant(L->suivant)
```

Question 10. *Écrire une procédure récursive qui prend en argument une liste* `L` *et retourne une nouvelle liste formée des éléments de* `L` *de valeur paire, dans le même ordre. On ne modifiera pas la liste* `L`*.*

Voici une solution récursive. Cf TD 3 pour une procédure de copie itérative.

```
copiePairs (L : liste) : liste
    si L == None alors
        retourner None
    si L->valeur % 2 == 1 alors
        retourner copiePairs(L->suivant)
    nouvelle : liste ; nouvelle = Nouveau(noeud)
    nouvelle->valeur = L->valeur
    nouvelle->suivant = copiePairs(L->suivant)
    retourner nouvelle
```