Examen (2 heures)

- Les documents (cours, TD, TP) sont autorisés.
- Les quatre exercices sont indépendants.
- À la fin de l'énoncé, il y a des détails pratiques concernant le devoir à la maison à rendre en fin de semaine. Ces détails sont à lire après l'examen (ou pendant si vous vous ennuyez...).

Exercice 1. *Fonctions récursives*

Écrivez les fonctions suivantes sur les listes ou les arbres de manière récursive en n'utilisant que les primitives `liste(),tete(l),queue(l),vide(l)` et `cons(x,l)` pour les listes et `arbre(),racine(a), gauche(a),droit(a),vide(a)` et `cons(x,a1,a2)` pour les arbres :

- `range(i,j)` qui renvoie une liste contenant les entiers de i à j (i et j sont des entiers et on suppose que $i \leq j$) ;
- `permute(l)` qui prend en argument une liste et renvoie une copie de la liste dans laquelle les éléments ont été permutés deux à deux. Par exemple, sur l'entrée $1, 2, 3, 4, 5, 6$ la fonction renvoie $2, 1, 4, 3, 6, 5$ (on supposera que la liste en entrée est toujours de longueur paire) ;
- `complet(a)` qui teste si un arbre binaire est complet, c'est-à-dire que tous les niveaux sont pleins (chaque sommet interne a 2 fils et les feuilles sont toutes à la même profondeur) ;
- `court_chemin(a)` qui renvoie la longueur du plus court chemin de la racine à une feuille dans un arbre binaire ;
- `structure(a1,a2)` qui teste si deux arbres binaires ont la même structure, c'est-à-dire qu'ils sont identiques si l'on ignore les étiquettes des nœuds ;
- `profondeurs(a)` qui renvoie la somme des profondeurs de chacun des nœuds de l'arbre `a` (la racine est à la profondeur 0, les fils de la racine sont à la profondeur 1, les fils de ces fils à la profondeur 2, etc.) ;
- `nb_sommets(a)` qui renvoie le nombre de sommets d'un arbre binaire `a` ;
- `ieme_sommet(a, i)` qui renvoie la i-ème valeur contenue dans un arbre binaire de recherche (selon l'ordre usuel sur les étiquettes, et en utilisant le fait que l'arbre est un arbre binaire de recherche). On pourra utiliser la fonction `nb_sommets`.

R.

```
def range(i, j):
  if i > j:
    return liste()
  else:
    return cons(i, range(i+1, j)

def permute(l):
  if vide(l):
    return liste()
  else:
    return cons(tete(queue(l)), cons(tete(l), queue(queue(l))))

def prof(a):
  if vide(a):
    return 0
  else:
    return 1 + max(prof(gauche(a)), prof(droit(a)))
def complet(a):
  """On utilise le fait qu'un arbre est complet si pour tous ses noeuds
  la profondeur du sous-arbre droit est égale à la profondeur du
```

```
    sous-arbre gauche (se prouve par récurrence)"""
    if vide(a):
        return True
    elif prof(gauche(a)) == prof(droit(a)):
        return complet(gauche(a)) and complet(droit(a))
    else:
        return False

def court_chemin(a):
    """Cette fonction doit chercher une feuille, pas uniquement un
    sous-arbre vide (pour éviter de s'arrêter au niveau d'un noeud n'ayant
    qu'un fils)."""
    if vide(gauche(a)) and vide(droit(a)):
        # on est sur une feuille
        return 0
    else:
        return 1 + min(court_chemin(gauche(a)), court_chemin(droit(a)))

def structure(a1, a2):
    if vide(a1):
        return vide(a2)
    elif vide(a2):
        return False
    else:
        return structure(gauche(a1), gauche(a2)) and structure(droit(a1), droit(a2))

def profondeurs(a):
    """On utilise une fonction annexe qui se souvient de la profondeur du
    noeud actuel"""
    def aux(a, p):
        if vide(a):
            return 0
        else:
            return p + aux(gauche(a), p+1), aux(droit(a), p+1)
    return aux(a, 0)

def nb_sommets(a):
    if vide(a):
        return 0
    else:
        return 1 + nb_sommets(gauche(a)) + nb_sommets(droit(a))

def ieme_sommet(a, i):
    nb_g = nb_sommets(gauche(a))
    if i <= nb_g:
        # le i-eme sommet est dans le sous-arbre gauche
        return ieme_sommet(gauche(a), i)
    elif i == nb_g + 1:
        # la racine est le i-eme sommet
        return racine(a)
    else:
        # le i-eme sommet est dans le sous-arbre droit
        return ieme_sommet(droit(a), i - nb_g - 1)
```

Exercice 2. *Retournement de listes*

1. Écrivez les fonctions suivantes de manière récursive (ici encore en utilisant les primitives de base sur les listes) :
- `append(a,l)` qui ajoute l'élément `a` à la fin de la liste `l` ;
- `concat(l1,l2)` qui concatène les listes `l1` et `l2` (les éléments de `l1` suivis des éléments de `l2` dans une même liste) ;
- `reverse(l)` qui renvoie la liste `l` retournée (premiers éléments à la fin).

R.

```
def append(a, l):
  if vide(l):
    return cons(a, liste())
  else:
    return cons(tete(l), append(a, queue(l)))
def concat(l1, l2):
  if vide(l):
    return l2
  else:
    return cons(tete(l), concat(queue(l1), l2))
def reverse(l):
  if vide(l):
    return l
  else:
    return concat(reverse(queue(l)), cons(tete(l), liste()))
```

2. Quelles sont les complexités des fonctions `append(a,l)` et `reverse(l)` en fonction de la longueur de la liste ?

R. La fonctions `append` parcourt une fois la liste, elle est donc de complexité $O(n)$. La fonction `reverse` appelle la fonction `append` pour chaque nœud de la liste, on appelle donc n fois la fonction `append`. La fonction `reverse` est donc de complexité $O(n^2)$.

On se propose d'améliorer la version récursive de `reverse(l)` en définissant une fonction `reverse_concat(l1,l2)` qui renvoie la concaténation de `reverse(l1)` et `l2` (par exemple, sur les entrées 1, 2, 3 et 4, 5, 6, la fonction `reverse_concat` doit renvoyer la liste 3, 2, 1, 4, 5, 6).

3. Écrivez directement la fonction `reverse_concat(l1,l2)` de manière récursive sans utiliser les fonctions `concat(l1,l2)` et `reverse(l)`.
Indication : il est recommandé de faire un schéma pour voir ce qui se passe et comprendre pourquoi cette fonction est facile à écrire.

4. En utilisant la fonction `reverse_concat(l1, l2)`, ré-écrivez la fonction `reverse(l)` (en deux lignes).

R.

```
def reverse_concat(l1, l2):
  if vide(l1):
    return l2
  else:
    return reverse_concat(queue(l1), cons(tete(l1), l2))
```

5. Quelle est la complexité de la nouvelle fonction `reverse(l)` en fonction de la longueur de la liste ?

R. Cette fonction ne parcourt qu'une seule fois la liste, elle est de complexité $O(n)$.

Exercice 3. *Expressions arithmétiques*

On considère les expressions arithmétiques sur les entiers n'utilisant que les opérateurs $+$, $-$, $\times$ et $\div$. Ces expressions peuvent être représentées par des arbres binaires dont les nœuds internes (nœuds non vides qui ne sont pas des feuilles) sont étiquetés par l'un des quatre opérateurs tandis que les feuilles sont étiquetées par des entiers.
Par exemple, l'arbre représenté sur la figure 1 représente l'expression arithmétique $(3-2)\times(7+10\div 2)$.

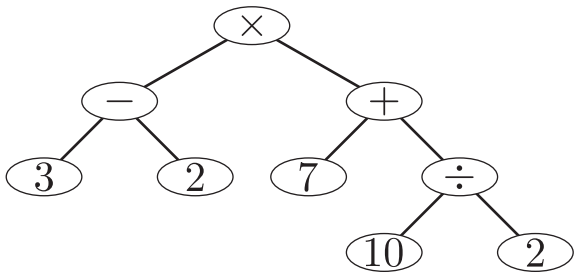

FIGURE 1 – Représentation de l'expression $(3-2)\times(7+10\div 2)$ par un arbre binaire.

Remarque : Les opérateurs sont représentés par la chaîne de caractères qui leur correspond. Par exemple, si l'on veut tester si un nœud contient l'opérateur + on pourra écrire `if n.valeur == '+'`.

1. En utilisant les classes `Noeud` et `Arbre` vues en cours (donc en manipulant directement les champs `gauche`, `droit`, `valeur` et `racine`), écrivez la fonction `valide(a)` qui teste si un arbre binaire `a` ne contenant que des entiers et des opérateurs représente bien une expression arithmétique valide (chaque nœud étiqueté par un opérateur doit avoir deux fils non vides et les entiers ne doivent apparaître que sur les feuilles de l'arbre).
Indication : On peut vérifier que la condition est vraie pour la racine puis tester récursivement si les sous-arbres droit et gauche sont également valides.

R.

```
def valide(a):
  r = a.racine  # r est un noeud
  if r == None:
    return True
  elif r.valeur in {'+', '-', '*', '/'}:
    return valide(arbre(r.gauche)) and valide(arbre(r.droit))
  else:
    return r.droit == None and r.gauche == None
```

2. Écrivez récursivement la fonction `eval(a)` qui renvoie la valeur correspondant à l'expression arithmétique représentée par l'arbre `a` (par exemple, sur l'arbre illustré par la figure 1, la fonction `eval` doit renvoyer 12). On suppose ici que l'arbre représente une expression valide.
Indication : Si le nœud considéré contient un entier, l'évaluation en ce nœud doit renvoyer la valeur de l'entier, si par contre le nœud contient par exemple un $+$, l'évaluation doit renvoyer le résultat de l'évaluation du fils gauche plus le résultat de l'évaluation du fils droit...

R.

```
def eval(a):
  r = a.racine
  if r.valeur == '+':
    return eval(arbre(r.gauche)) + eval(arbre(r.droit))
  elif r.valeur == '*':
    return eval(arbre(r.gauche)) * eval(arbre(r.droit))
```

```
    elif r.valeur == '/':
      return eval(arbre(r.gauche)) / eval(arbre(r.droit))
    elif r.valeur == '-':
      return eval(arbre(r.gauche)) - eval(arbre(r.droit))
    else:
      return r.valeur
```

Exercice 4. *Langages et arbres ternaires*

On va ici étudier une structure de données permettant de stocker un ensemble fini de mots (un langage) de telle sorte qu'il soit facile d'ajouter, supprimer et rechercher des mots.
Pour cela, on utilise des arbres « ternaires » c'est-à-dire des arbres dans lesquels chaque nœud non vide a 3 fils : gauche, centre et droit.
Afin de stocker des mots, les nœuds des arbres considérés sont étiquetés par des lettres. De plus, pour des raisons techniques qui seront expliquées plus tard, il faut ajouter un champ `terminal` aux nœuds contenant un booléen (vrai ou faux) pour indiquer les nœuds qui correspondent à la fin d'un mot du langage. On utilise donc les deux classes suivantes :

```
class Noeud:
    pass
def noeud(x):
    n = Noeud()
    n.lettre = x
    n.terminal = False
    n.gauche = None
    n.centre = None
    n.droit = None

class Arbre:
    pass
def arbre(n = None):
    a = Arbre()
    a.racine = n
```

Le premier mot que l'on insère dans l'arbre est inséré « verticalement » c'est-à-dire que sa première lettre est placée sur la racine puis chacune des lettres suivantes est sur le fils centre de la précédente (la figure 2 illustre un exemple). Lorsque l'on veut insérer un nouveau mot dans l'arbre (dans l'exemple de la figure, le mot *metre* après le mot *mars*), on part de la racine puis on descend (fils centre) tant que l'on trouve des lettres correspondant à celles du mot à insérer (ici la lettre *m* est correcte, mais la lettre suivante *a* n'est pas bonne).
Lorsqu'une lettre ne correspond pas, on part alors vers le fils gauche ou droit selon que la lettre à insérer est plus petite (gauche) ou plus grande (droit) que la lettre courante dans l'arbre. Lorsque l'on tombe sur la lettre qu'on voulait insérer on recommence à descendre selon le fils centre (si on doit insérer une lettre à droite d'un nœud et que ce nœud n'a pas de fils droit, on en créé un nouveau qui contient la lettre que l'on veut et l'on ajoute les lettres suivantes en dessous).
Lorsque l'on a inséré toutes les lettres du mot, on marque le nœud contenant la dernière lettre comme étant terminal (son champ `terminal` devient `True`).
La dernière étape de la figure 2 montre l'arbre obtenu après insertion successive de huit mots (et il est probablement plus facile de comprendre le fonctionnement en observant la figure qu'en lisant l'explication).
L'intérêt de ces arbres est qu'il faut peu d'espace pour stocker des mots ayant des préfixes communs. Chaque chemin dans l'arbre de la racine vers un nœud terminal correspond à un mot du langage, qui est obtenu en ne lisant que les lettres sur lesquelles dont on a suivi le fils centre (les lettres du chemin pour lesquelles on sort par le fils gauche ou droit sont celles qui n'étaient pas correctes pour notre mot).

1. Dessinez l'arbre ternaire obtenu après insertion des mots suivants (à partir d'un arbre initialement vide) : *chien*, *chat*, *cheval*, *chevaux*, *castor*, *chaton*, *choux* et *chou*.

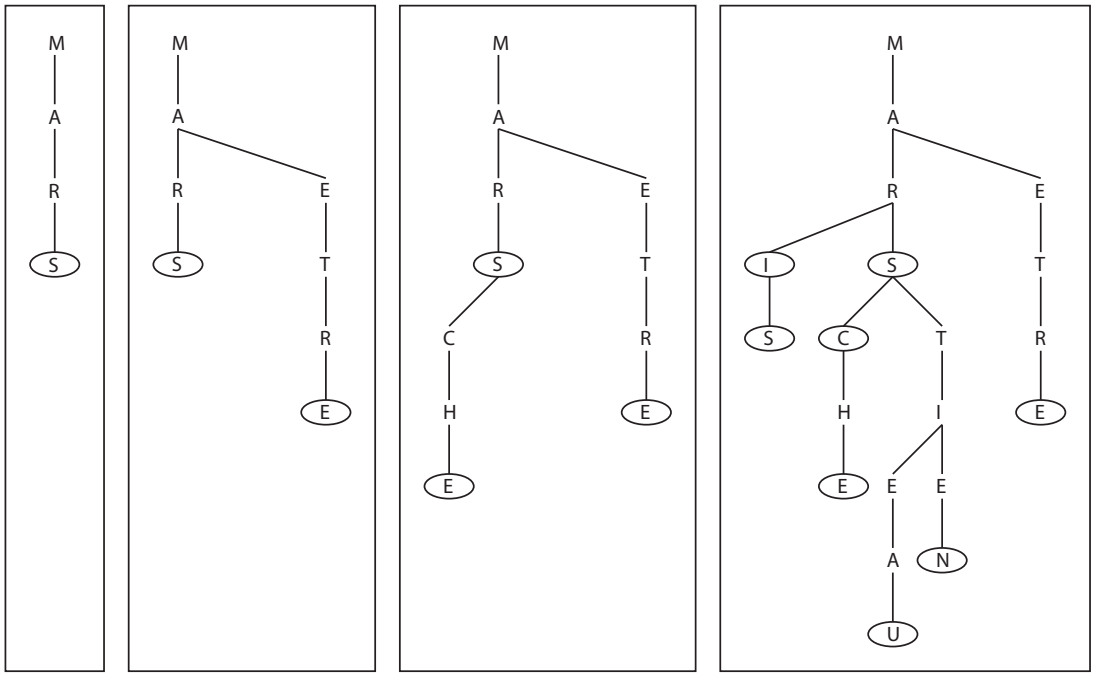

FIGURE 2 – Les différents stades de l'arbre après insertion successive des mots *mars*, *metre*, *marche* puis finalement *mai*, *marc*, *mais*, *martien* et *marteau*. Les nœuds entourés sont les nœuds terminaux.

L'insertion dans un arbre ternaire peut être réalisée par la fonction suivante (la fonction a l'air impressionnante parce qu'elle est très lourdement commentée, en réalité il n'y a qu'une vingtaine de lignes de véritable code *Python* que vous pouvez recopier sur un brouillon si ça vous aide à mieux comprendre) :

```
def insere(w, a):
  if a.racine == None:
  # si l'arbre est vide, il faut créer un premier noeud qui contient
  # la première lettre du mot à insérer
    n = noeud(w[0])
    a.racine = n

  # On va ensuite utiliser une fonction récursive annexe qui prend en
  # argument un mot et un noeud et insere le mot à partir de ce noeud
  # (on va donc pouvoir descendre dans l'arbre en rappelant cette même
  # fonction sur un noeud plus bas)
  def insere_rec(w, n):
    if w[0] == n.lettre:
    # la première lettre du mot coincide avec la lettre sur le
    # noeud considéré : on n'a donc pas de noeud à rajouter, on
    # se contente de descendre dans l'arbre (selon le fils centre)
    # et on regarde la lettre suivante du mot w.
      if len(w) == 1:
      # si le mot w n'a plus de lettre après celle-ci, il suffit
      # d'indiquer que le noeud que l'on vient d'atteindre est
      # est la fin d'un mot du langage
        n.terminal = True
      else:
        if n.centre == None:
```

```
        # si le noeud n'a pas de fils centre, il faut en créer
        # un avant de descendre
          n1 = noeud(w[1])
          n.centre = n1
        insere_rec(w[1:], n.centre)
        # on rappelle la fonction récursivement, en descendant
        # dans l'arbre et en avançant dans le mot
        # w[1:] désigne le mot w privé de sa première lettre
    elif w[0] < n.lettre:
    # si la première lettre du mot est plus petite que la lettre
    # sur le noeud courant, il va falloir insérer le mot à gauche
    # du noeud.
      if n.gauche == None:
      # si le noeud n'a pas de fils gauche, il faut en créer un
        n1 = noeud(w[0])
        n.gauche = n1
      insere_rec(w, n.gauche)
      # on insère récursivement le mot à gauche du noeud.
    else:
    # cas symétrique du précédent, mais on insère à droite.
      if n.droit == None:
        n1 = noeud(w[0])
        n.droit = n1
      insere_rec(w, n.droit)
# Fin de la définition de la fonction récursive annexe.
insere_rec(w, a.racine)
# on appelle la fonction récursive annexe avec comme premier
# noeud la racine de l'arbre.
```

2. Écrivez la fonction `recherche(w,a)` qui teste si un mot `w` donné appartient au langage représenté par l'arbre ternaire `a` (il faut parcourir l'arbre en fonction des lettres du mot puis vérifier que le nœud sur lequel on arrive est bien terminal).

R.

```
def recherche(w, a):
  r = a.racine
  if r == None:
    return False
  elif r.valeur == w[0]:
    if len(w) == 1:
      return r.terminal
    else:
      return recherche(w[1:], arbre(r.centre))
  elif r.valeur > w[0]:
    return recherche(w, arbre(r.gauche))
  else:
    return recherche(w, arbre(r.droit))
```

3. Écrivez la fonction `supprime(w,a)` qui modifie l'arbre `a` pour que le mot `w` n'appartienne plus au langage qu'il représente.

Remarque : Dans un premier temps, on peut se contenter de rendre le nœud correspondant à la dernière lettre de `w` non terminal. Cependant si l'on ne fait que ça, l'arbre finit par contenir plusieurs branches qui ne servent à rien parce qu'elles ne contiennent plus aucun nœud terminal (il est donc inutile de les garder puisqu'elles ne représentent aucun mot du langage). Il faut alors faire en sorte que la fonction de suppression élimine ces branches.

R. Voici la solution simple (on se contente de changer le champ `terminal` du nœud correspondant à la fin du mot). La version complète où l'on supprime les branches restantes est nettement plus pénible à écrire (et n'était dans le sujet d'examen que pour occuper au cas où quelqu'un finirait en avance).

```
def supprime(w, a):
  r = a.racine
  if r == None:
    # le mot n'est pas dans l'arbre, il n'y a rien à supprimer
    return
  elif r.valeur == w[0]:
    if len(w) == 1:
      r.terminal = False
    else:
      supprime(w[1:], arbre(r.centre))
  elif r.valeur > w[0]:
    supprime(w, arbre(r.gauche))
  else:
    supprime(w, arbre(r.droit))
```

À propos du devoir à la maison

- Le devoir à la maison est à rendre par mail (à l'adresse `victor.poupet@lif.univ-mrs.fr`) avant samedi 8 janvier à minuit ;
- Il y aura une pénalité de 2 points par heure de retard arrondie au dessus (l'heure du serveur mail faisant foi). Il est donc fortement déconseillé de jouer à envoyer le mail à minuit pile ;
- Vous devez impérativement joindre un fichier texte (idéalement avec l'extension `.py`) contenant le code des fonctions ;
- Pour ce qui est des réponses aux questions qui demandent des justifications (preuves, ou complexités de fonctions) vous pouvez soit les insérer directement dans le fichier `.py` comme commentaires (en ajoutant un symbole # en début de ligne) soit répondre à ces questions dans le mail ;
- Vous devez tester les fonctions avant de les envoyer. Dans ce devoir, vous devez au moins vérifier que si l'on créé un tas et que l'on ajoute plusieurs valeurs avec la fonction `push` puis que l'on exécute la fonction `pop`, non seulement les fonctions ne génèrent pas d'erreurs mais en plus elles renvoient bien les valeurs qui avaient été insérées. Cette remarque est également valable pour les gens qui ont déjà envoyé leur devoir, car il est évident que certains n'ont rien testé du tout ;
- Pour information, le devoir de cette année porte sur les files de priorités et les tas. Le devoir à la maison concernant les intervalles était celui de l'an dernier...
- Le DM compte pour 1/4 de la note finale quoi qu'il arrive (pas seulement si la note est meilleure que celle de l'examen).