# Composition d'Informatique

# Les Principes des Langages de Programmation (INF 321)

Promotion 2007

Sujet proposé par Gilles Dowek

7 juillet 2008

Les exercices qui suivent sont indépendants et peuvent être traités dans n'importe quel ordre. On attachera une grande importance à la clarté, à la précision et à la concision de la rédaction.

## Exercice 1 (2 points)

On considère des entiers représentés en base dix par des listes de chiffres, chaque chiffre étant de type `int` et le chiffre de poids faible étant en tête de la liste.

1. Écrire une fonction qui ajoute `1` à un entier ainsi représenté.
2. Écrire une fonction qui ajoute deux entiers ainsi représentés.

## Exercice 2 (5 points)

Pour chaque entier `n`, on se donne deux symboles, la parenthèse ouvrante $(_n$ et la parenthèse fermante $)_n$, et on appelle *mot* une suite finie formée de tels symboles.

L'ensemble des mots *bien formés* est défini par les trois règles suivantes :

– le mot vide est bien formé,

– si `w` est un mot bien formé, alors $(_n w)_n$ est un mot bien formé,

– si `w` et `w'` sont des mots bien formés alors, `ww'` est un mot bien formé.

1. Le mot $(_0(_1)_1)_0(_1)_1$ est-il bien formé ?
2. Et le mot $(_1(_0)_0)_0(_1)_1$ ?
3. On veut écrire une fonction qui vérifie qu'un mot est bien formé. Pour cela, on parcourt le mot de gauche à droite. Quand on rencontre un symbole $(_n$, on empile l'entier `n`, et quand on rencontre un symbole $)_n$, on vérifie que l'entier au sommet de la pile est `n` et on le dépile.

Montrer que si un mot est bien formé, alors si on commence son parcours avec une pile dans un état `p`, on achève ce parcours avec une pile dans le même état `p`. En déduire que si un mot est bien formé, alors en commençant son parcours avec une pile vide on achève ce parcours avec une pile vide également. (On admettra le réciproque).

4. Définir le type des mots.
5. Écrire une fonction qui vérifie qu'un mot est bien formé.

## Exercice 3 (5 points)

Une liste d'entiers est dite *croissante* si chaque entier de la liste est inférieur ou égal à l'élément qui le suit dans la liste.

1. Écrire une fonction qui imprime toutes les listes croissantes de `p` entiers compris entre `0` et `m`.

2. Montrer que le nombre de listes affichées est égal à `(p + m)!/(p! m!)`.

## Exercice 4 (5 points)

On considère le type des arbres binaires avec un contenu entier.

```
class Arbre {
 int val;
 Arbre gauche;
 Arbre droite;

 Arbre (int v, Arbre g, Arbre d) {val = v; gauche = g; droite = d;}}
```

Et une la méthode ainsi définie

```
 static boolean parcours (Arbre a) {
   if (a == null) return false;
   return (a.val == 0) || parcours(a.gauche) || parcours(a.droite);}
```

On définit trois arbres ainsi

```
Arbre a = new Arbre (1,new Arbre (2,null,null), new Arbre (3, null,null));
Arbre b = new Arbre (1,new Arbre (0,null,null), new Arbre (3, null,null));
Arbre c = new Arbre (1,new Arbre (2,null,null), new Arbre (0, null,null));
c.gauche.droite = c;
```

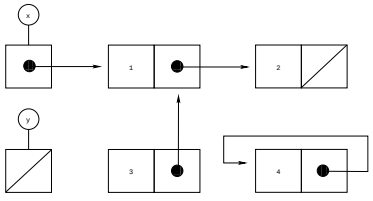
1. Dessiner ces trois arbres

2. Que retournent les appels `parcours(a)`, `parcours(b)` et `parcours(c)` ?

3. Combien de nœuds sont visités dans chacun de ces cas ?

4. Comment modifier la fonction `parcours` pour qu'elle retourne `true` ou `false` selon que l'arbre contient un `0` ou non, y compris dans le cas d'arbres infinis rationnels ? (On pourra dans cette fonction utiliser un type `ListArbres` des listes d'arbres et la méthode `mem` sur ces listes.)

## Exercice 5 (3 points)




1. On effectue un glanage de cellules (gc) dans l'état ci-dessus en utilisant la méthode du marquage. Dessiner l'état à l'issue de ce glanage de cellules.