TP 3 (partie A)

N'oubliez pas de remplacer les noms fictifs par vos noms dans chacun des gabarits.

1. Ce numéro vise à bien saisir les différences entre let, let* et letrec. (voir les diapositives du thèmes 2, plus particulièrement celle à partir de 62.
   Sauf pour l'utilisation de let, let* et letrec, les trois expressions suivantes sont identiques. Récrivez ces trois expressions, en éliminant les let, let* et letrec.

a)
```
(let ([f (λ (u v w)
           (g u v w))]
      [g (λ (x y z)
           (cond [zz
                  (set! zz (not zz))
                  (f x y '(zz))]
                 [else
                  'ouf]))])
  (list (f '(i)'(j)'(k)) (g '(e)'(f)'(g))))
```

b)
```
(let* ([f (λ (u v w)
           (g u v w))]
      [g (λ (x y z)
           (cond [zz
                  (set! zz (not zz))
                  (f x y '(zz))]
                 [else
                  'ouf]))])
  (list (f '(i)'(j)'(k)) (g '(e)'(f)'(g))))
```

c)
```
(letrec ([f (λ (u v w)
              (g u v w))]
         [g (λ (x y z)
              (cond [zz
                     (set! zz (not zz))
                     (f x y '(zz))]
                    [else
                     'ouf]))])
  (list (f '(i)'(j)'(k)) (g '(e)'(f)'(g))))
```

Les trois expressions ne sont pas nécessairement exécutables parce que f g ou zz doivent être définis avant leurs utilisations. Vous pouvez utiliser les définitions ci-contre pour tester vos solutions, elles sont déjà

```
(define f (λ (a b c) (list a b c)))
(define g (λ (a b c) (append a b c)))
(define zz #t)
```

incluses dans le gabarit «no 1 transformer let en lambda.rkt» que vous devez utiliser pour remettre votre réponse. .

2. Redéfinir la fonction flatten vu en utilisant les fonctions d'ordre supérieure.
   Vous devez utiliser le gabarit intitulé «*no2 flatten.rkt*».

   a. la en utilisant append-map, cette version de flatten est nommé flatten1

   b. en utilisant foldr, cette version de flatten est nommé flatten2

3. Définir la fonction map-arbre, une version de map pour les arbres, en utilisant judicieusement la fonction map. Voici deux exemples d'utilisation de map-arbre.

```
(map-arbre sqr '(1 (2 (3)) () 4 5))
==> '(1 (4 (9)) () 16 25)

(map-arbre number->string '(1 (2 (3)) () 4 5))
==> '("1" ("2" ("3")) () "4" "5")
```

   Vous devez utiliser le gabarit intitulé «*no3 map-arbre.rkt*»

4. Définir produit-fonctionnel-a et produit-fonctionnel-b qui implémentent le produit fonctionnel. Cette exercice a pour but comme les précédents d'acquérir une meilleure maîtrise des fonctions d'ordre supérieure mais aussi de choisir la forme syntaxique appropriée de λ (voir la diapositive 13 du thème 5). Vous devez utiliser le gabarit intitulé «*no4 produit-fonctionnel.rkt*»

   a. produit-fonctionnel-a effectue la composition d'une liste de fonctions donnée comme argument. Vous devez utiliser curry et foldr ou foldl de façon pertinente. La sémantique de produit-fonctionnel est ((produit-fonctionnel-a (f_0 f_1… f_n)) x) -> (f_0 (f_1 … (f_n x))).

   b. produit-fonctionnel-a effectue la composition d'un nombre arbitraire de fonctions données en arguments. Vous devez utiliser foldr ou foldl de façon pertinente. La sémantique de produit-fonctionnel est ((produit-fonctionnel-a f_0 f_1… f_n) x) -> (f_0 (f_1 … (f_n x)).

   Comme l'illustre l'encadré ci-dessous, les deux versions donnent la même réponse, c'est simplement la façon de donner les fonctions à composer qui changent.

```
((produit-fonctionnel-a (list add1 sqr (curry * 2))) 1)
==> 5

((produit-fonctionnel-b add1 sqr (curry * 2)) 1)
==> 5
```