TP 3 (partie B)

Cette partie du TP3 est à remettre sur Moodle avant 22:00, le vendredi 3 novembre.
Vous devez utiliser les 3 gabarits suivants pour remettre votre réponse.
no 5 commuter_ij.rkt ;
no 6 table de vérité.rkt
no 7 produit-cartésien.rkt .
Vous devez aussi utiliser le module 3 comme gabarit pour le no 8.
Évidemment au début de chaque fichier, vous devez remplacer les noms fictifs par vos noms.
L'exécution de vos fichiers ne doit pas retourner de valeur ou afficher des messages, i.e. enlever tous les messages ou tests que vous avez inclus dans votre fichier.

5. Définissez en DrRacket la fonction commuter_ij. Cette fonction prend une liste et deux entiers et retourne une liste identique sauf que les éléments i et j sont permutés. Les indices commencent à zéro. Vous pouvez assumer que i < j < (length liste). Voici un exemple
(commuter_ij '((λ (x y) y x) 9 (5 6) (+ a b) 4)1 3) retourne '((λ (x y) y x) (+ a b) (5 6) 9 4). Pour définir cette fonction, n'utilisez que les primitives suivantes : cons car cdr caar cadr … append reverse let let* letrec cond if list + - ... < > = … null? list? pair? Cependant, si vous croyez qu'une autre primitive est utile, vous pouvez la redéfinir.
Important : commuter_ij ne doit traverser la liste qu'une fois pour identifier les différentes parties de l'expression et une autre fois seulement pour reconstruire l'expression à partir des parties.

6. Définissez produit-cartésien. Produit cartésien prend un nombre arbitraire de liste comme argument. Voici quelques exemples d'évaluation de produit-cartésien :

```
(produit-cartésien) ⇨
(())

(produit-cartésien '(X Y Z))⇨
((X) (Y) (Z))

(produit-cartésien '(* - +) '(1 2 3))⇨
((* 1) (* 2) (* 3) (- 1) (- 2) (- 3) (+ 1) (+ 2) (+ 3))

(produit-cartésien '((A a) (B b) (C c)) '(1 2 3))⇨
(((A a) 1) ((A a) 2) ((A a) 3)
 ((B b) 1) ((B b) 2) ((B b) 3)
 ((C c) 1) ((C c) 2) ((C c) 3))

(produit-cartésien '(X Y Z) '(1 2 3) '(a b c))⇨
((X 1 a) (X 1 b) (X 1 c) (X 2 a) (X 2 b) (X 2 c) (X 3 a) (X 3 b) (X 3 c)
 (Y 1 a) (Y 1 b) (Y 1 c) (Y 2 a) (Y 2 b) (Y 2 c) (Y 3 a) (Y 3 b) (Y 3 c)
 (Z 1 a) (Z 1 b) (Z 1 c) (Z 2 a) (Z 2 b) (Z 2 c) (Z 3 a) (Z 3 b) (Z 3 c))
```

7. Logique des propositions et table de vérité.
L'objectif est de construire des tables de vérité en mesure de déterminer si une expression logique est une tautologie, une contradictions ou ni l'un ni l'autre.
Les expressions logiques sont construites à partir des opérateurs : ¬, ∧ et ∨.

```
(define ¬ not)
(define ∧ (λ (a b)(and a b)))
(define ∨  (λ (a b)(or a b)))
```

a. Définissez liste-de-cas. liste-de-cas prend n, un entier positif, et retourne une liste dont chaque élément est une liste correspondant à une ligne d'une table de vérité pour une expression à n arguments (sauf celle de la valeur de l'expression elle-même).
Important : liste-de-cas doit avoir une forme récursive terminale pour s'exécuter de façon itérative.
Par exemple :

```
(liste-de-cas 2)
'((#t #t)
  (#f #t)
  (#t #f)
  (#f #f))
```

```
(liste-de-cas 3)
'((#t #t #t)
  (#f #t #t)
  (#t #f #t)
  (#f #f #t)
  (#t #t #f)
  (#f #t #f)
  (#t #f #f)
  (#f #f #f))
```

b. Définissez table-de-vérité. table-de-vérité prend une expression logique sous forme d'une fonction, affiche la table-de-vérité complète, et retourne l'un des symboles suivants : 'tautologie, 'contradiction, 'aucun-des-deux.
Définissez aussi map-ajoute-à-la-fin, une fonction illustrée par l'interaction suivante :
(map-ajoute-à-la-fin '((a b) (c d) (e f)) '(1 2 3))

'((a b 1) (c d 2) (e f 3))
Exemple

(table-de-vérité ∧) (#t #t #t) (#f #t #f) (#t #f #f) (#f #f #f) 'aucune-des-deux	(define resol (λ (a b c) (and (or a b) (or c (not b))))) (define expr1 (λ (a b c) (↔ (resol a b c) (or a c)))) (table-de-vérité expr1) (#t #t #t #t) (#f #t #t #t) (#t #f #t #t) (#f #f #t #t) (#t #t #f #t) (#f #t #f #t) (#t #f #f #t) (#f #f #f #t) 'tautologie	(table-de-vérité (λ (a b c d) (∧ (∧ (∨ a (¬ b)) (∨ (¬ c) d)) (∧ (¬ a) b)))) (#t #t #t #t #f) (#f #t #t #t #f) (#t #f #t #t #f) (#f #f #t #t #f) (#t #t #f #t #f) (#f #t #f #t #f) (#t #f #f #t #f) (#f #f #f #t #f) (#t #t #t #f #f) (#f #t #t #f #f) (#t #f #t #f #f) (#f #f #t #f #f) (#t #t #f #f #f) (#f #t #f #f #f) (#t #f #f #f #f) (#f #f #f #f #f) 'contradiction

8. 3 méta-évaluateur applicatif.rkt, le gabarit de ce numéro contient tout le code de «3 méta-évaluateur applicatif.rkt» vu en classe. Vous devez le modifier pour implémenter letrec. Voici une description du comportement du letrec
(letrec ([id val-expr] ...) body ...+)
   a. un nouvel environnement est attaché à l'environnement courant ;
   b. tous les id sont ajoutés à ce nouvel environnement et leurs valeurs est undefined ;
   c. dans l'ordre, chaque val-expr est évalué dans ce nouvel environnement et immédiatement lié à id correspondant ;
   d. la séquence body...+ est évalué et la dernière évaluation est retournée.

   Le fichier trace que votre implémentation crée doit être identique (ou presque) pour le programme «progr-letrec.txt» au fichier «trace progr-letrec.txt» et pour le programme «progr-letrec.1.txt» au fichier «trace progr-letrec.1.txt» .

   Remarque 1 : Vous implémentation doit se faire uniquement en modifiant le fichier «3 méta-évaluateur applicatif.rkt». Il est cependant permis d'ajuster les identificateurs path-progr et path-trace en fonction de votre installation.

   Remarque 2 : Pour ce numéro vous ne remettez que le gabarit que vous aurez modifier pour intégrer votre implémentation du letrec.

   Voici cependant les liens vers les trois autres fichiers :
   1 initialisation.rkt
   2 structures de données - constructeur accesseur mutateur.rkt

4 REPL.rkt