TP 4

Le TP est à remettre sur Moodle avant 22:00, le dimanche le 12 novembre.
Le TP se divise en trois parties: la première est qui vise à mieux comprendre l'évaluation par environnement et le méta-évaluateur applicatif ; la seconde vise à améliorer votre maîtrise de l'appariement de forme ; et la troisième porte sur la création de nouvelle formes syntaxiques.

Partie 1 (9 points)

1. 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

Partie 2 (24 points)

2. Concevoir i->p, la fonction qui transforme une expression infixée en une expression préfixée pour un domaine défini à l'aide du fichier commun.rkt. Ce fichier définit les constantes du domaine, les variables ainsi que les opérateurs. Il spécifie la priorité des opérateurs et il spécifie ceux qui sont unaires, binaires, les autres étant n-aires. L'opérateur qui a la priorité la plus forte est associé à l'entier le plus petit et c'est lui qui prend les opérandes ; par exemple * a une priorité plus forte que + donc (< (priorité *) (priorité +)), et (2 + 3 * 4 + 5) = (2 + (3 * 4) + 5).

   a. Votre code doit être indépendant des opérateurs définis dans le fichier commun.rkt, i.e. lors de la correction, les tests seront faits avec un fichier "commun.rkt" différent.

   b. Voici le gabarit pour cette partie i2p.rkt .

c. L’exécution de vos fichiers ne doivent rien afficher ou retourner.

d. Comme il ne semble pas avoir de convention bien établi sur les priorités relatives des opérateurs, vous pouvez assumer les faits suivants :

e.

f. Si 2 opérateurs n-aires ont la même priorité alors l'association se fait à gauche ; par exemple si # et & ont la même priorité alors (a # b & c) = ((a # b) & c)

g. Si ¤ est un opérateur binaire, # est un opérateur n-aire et qu’ils ont tous deux la même priorité alors la priorité est donné à l’opérateur binaire ; par exemple (a # b ¤ c) = (a # (b ¤ c))

h. Si un opérateur est défini comme binaire alors l'association se fait aussi à gauche ; par exemple si ¤ est binaire alors (a ¤ b ¤ c) = ((a ¤ b) ¤ c)

i. Si un opérateur est défini comme unaire alors il a la priorité la plus forte et il précède toujours son opérande ; par exemple # est unaire alors (# a & # # b) = ((# a) & (# (# b)))

j. Les expressions en notation préfixée sont complètement parenthésées, i.e si # est binaire vous ne pouvez pas écrire (# a b c) pour dire (# (# a b) c) ; et dans le cas unaire, vous ne pouvez pas écrire (# # b) à la place de (# (# b))

k. Les parenthèses servent uniquement à outrepasser la priorité des opérateurs. Vous pouvez aussi assumer qu'il n'y a pas de parenthèses inutiles, i.e. encadrant un seul atome, par exemple (a), ni de double parenthèses, par exemple ((# a)), ((a ¤ b)), ni des parenthèses qui sont redondantes avec la priorité des opérateurs, par exemple ((a ¤ b) ¤ c). Cependant pour les opérateurs unaires dans la forme infixée vous pouvez avoir aussi bien (# # (a ¤ b)) que (# (# (a ¤ b)))

l. L’objectif de cette seconde partie est d’augmenter votre maîtrise de l’appariement de forme à l’aide de la librairie match. Il faut donc utiliser judicieusement cette librairie pour faire de l’appariement de forme pour réussir cette partie du TP. Je suggère en particulier que vous utilisiez parfois match*, cette fonction est identique à match mais permet de faire de l’appariement sur plusieurs expressions en parallèle. Plusieurs conception de i->p sont possibles, je vous suggère la mienne. L’expression est traversée de gauche à droite et j’identifie argg, le premier opérande. Comme argg peut être une expression unaire, une expression parenthésée ou un simple élément, j’ai conçu des fonctions spécifiques identifier la situation et construire ce premier opérande. op est l’élément qui suit et est nécessairement une opérande. J’identifie ensuite argd, le prochain opérande, le reste de l’expression est nommé reste. Le plus gros du traitement est fait par une fonction nommée traiter-n-aire, il prend 4 arguments : argg, op, argd et reste. Avec les trois premier, il construit le nouvel argg et utilise reste pour identifier le prochain opérateur, le nouvel argd et le nouveau reste, et se rappelle avec ces nouveaux arguments tant que reste n’est pas nul. Pour vous aider, je mets à votre disposition mon fichier de test et mon fichier et celui pour tagger les expressions qui sont déjà entre parenthèse dans l’expression initiale. Je conserve ces tags jusqu’à la fin du traitement et effectue une seconde passe sur la nouvelle expression pour les enlever. Il y a moyen d’éviter cette seconde passe mais ça allongerait le code sans augmenter de façon significative votre maîtrise de l’appariement de forme.

Partie 3 (15 points)

Vous devez coder les formes syntaxiques my-apply, one-before et ~~. Ces trois formes syntaxiques doivent être coder dans les gabarits correspondants : my-apply.rkt, one-before.rkt, quasi-quote.rkt.

3. my-apply est une forme syntaxique qui simule le apply de Racket (http://docs.racket-lang.org/reference/procedures.html) pour les cas où le dernier argument est de la forme '(...)

   Exemple

   (my-apply + 1 2 '(3 4 5)) ⇒ 15

   (my-apply * '(3 4 5)) ⇒ 60

   (my-apply + 1 (+ 1 5) '(4 5)) ⇒ 16

   Mais vous n'avez pas à traiter des situations similaires aux suivantes :

   (my-apply + 1 2 3 (map sqr '(3 4)))

   (my-apply * 1 (+ 1 5) (list 4 5))

4. one-before est une forme syntaxique qui ressemble à begin sauf qu'elle retourne l'avant-dernière expression. Si la séquence d'expression est vide, one-before retourne le message d'erreur suivant "one-before vide". Si la séquence d'expression ne contient qu'un élément alors cet élément est premièrement affiché et ensuite retourné.

   Exemple :

```
(one-before
  (displayln "ligne 1")
  (displayln "la prochaine ligne \"(sqrt 12345)\" est l'expression à retourner")
  (sqrt 12345)
  (displayln "la dernière ligne"))
⇒
ligne 1
la prochaine ligne "(sqrt 12345)" est l'expression à retourner
la dernière ligne
111.1080555135405
(one-before) Þ "one-before vide"
'(one-before
  ((λ () (displayln "Cette ligne doit être affichée qu'une fois.")
         (sqrt 12345))))
⇒
((λ () (displayln Cette ligne doit être affichée qu'une fois.) (sqrt 12345)))
Cette ligne doit être affichée qu'une fois.
111.1080555135405
(one-before (begin (+ 1 2 3 4) (* 1 2 3 4)))
⇒
(begin (+ 1 2 3 4) (* 1 2 3 4))
24
```

5. ~~ est une forme syntaxique identique (ou presque) quasiquote (https://docs.racket-lang.org/reference/quasiquote.html?q=quasiquote) qui permet facilite la création de liste à l'aide de deux littéraux ~ et ~@.
   - ~~ seul a le même comportement que quote (').
     Par exemple : (~~ a b (+ 1 2)) ⇒ '(a b (+ 1 2))
   - Si un élément de la liste est précédé du littéral ~ alors il est évalué avant d'être mis dans la liste.
     Par exemple :
     (define b 2)
     (define c 3)
     (~~ ~ b ~ c (a b c)) ⇒ '(2 3 (a b c))
     (~~ x b ~ b y ~ (+ b 5)) ⇒ '(x b 2 y 7)
   - Si un élément de la liste est précédé du littéral ~@ alors il est évalué et le résultat doit être une liste dont les éléments sont insérés dans la liste qui sera retournée par ~~.
     Par exemple
     (define ls (list '+ '* '&))
     (~~ ~@ ls) ⇒ '(+ * &)
     '(~~ 3 ~@ ls ~@ (range 5) 12) ⇒ '(3 + * & 0 1 2 3 4 12)