TD n°8

Ce TD est un bric à brac de choses vues pendant le trimestre
Il doit permettre de vérifier ce que l'on sait, mettre au point les idées, etc.

I Quelques utilitaires

1) Outil de trace (déjà donné dans le TD n°5)

En début de fenêtre de définition, écrire :

```
(require (lib "trace.ss"))
```

Puis après évaluation de la fenêtre de définitions, taper dans la fenêtre d'évaluation :

`(trace puis mul …)` ; remarque la fonction trace est une fonction n_aire

Pour arrêter la trace, il suffit de taper dans la fenêtre d'évaluation :

```
(untrace puis mul …)
```

2) Outils pour mesurer le temps d'exécution d'un fonction

Après évaluation de la fenêtre de définitions des fonctions que l'on vuet tester, taper dans la fenêtre d'évaluation :

(time (*nom_fonction arg1 arg2*…))

Exemple :

```
>(time (fac 20000))
cpu time: 1625 real time: 1625 gc time: 393

le temps réel de calcul est 1625 – 393 = 1232 ms
```

L'inconvénient de cet appel est qu'il vous retourne également le résultat du calcul de (fac 20000) qui est un nombre gigantesque… (voir exercices suivants)

Si on est juste intéressé par l'estimation du temps de calcul de la fonction mais pas par le résultat de la fonction, il suffit de taper :

```
time (void (fac 20000)))
cpu time: 3469 real time: 3469 gc time: 2218

le temps réel de calcul est 3469 – 2218 = 1251 ms
```

Remarque : le temps de calcul d'une fonction dépend évidemment de l'ordinateur. L'utilisation de la fonction time peut servir à comparer différentes fonctions sur la même machine

3) Outils pour passer entre types

DrRacket manipule des nombres, des symboles, des chaines de caractères, des vecteurs (comme des tableaux), etc. mais surtout il permet de manipuler des listes. Il peut être utile de transformer par exemple une chaine de caractères en listes (de caractères évidemment), un vecteur en liste , ou l'inverse. Il existe des fonctions qui permettent ce genre d'opérations.

Taper les instructions suivantes :

```
(define chaine "ceci est une chaine de caractères")
(define liste '(ceci n'est pas une chaine de caractères))  ; -attention au quote - si ce n'est pas une chaine mais alors qu'est ce que c'est ?
(define chaineenliste (string->list chaine))
(define listenenchaine (list->string chaineenliste))
```

Ne pas taper au clavier la fonction suivante et réfléchir à ce que l'évaluateur va retourner :

```
(define listenenchaine2 (list->string liste))
```

Puis la taper et voir si le résultat est conforme à vos prévisions !
Pourquoi l'évaluateur retourne-t-il un tel résultat ?

Taper ensuite les fonctions suivantes et essayer de prévoir le résultat avant de faire évaluer chaque fonction.

```
(define nombre 1515)
(define nombreenchaine (number->string nombre))
(define pieenchaine (number->string pi))    ; évidemment DrRacket connaît Pi
nombre
pi
nombreenchaine
pieenchaine
(+ pi nombre)
(+ pieenchaine  nombre)
(+  nombreenchaine pieenchaine)
(string-append nombreenchaine pieenchaine)
```

Bilan rapide :
Une chaine de caractère s'écrit entre guillemets
`Exemple "ceci est une chaine de caractères"`

Un caractère s'écrit sous la forme #\..
`Exemple : #\A mais attention #\A est différent de #\a`
Pour le vérifier, taper :

```
(eqv? (cadr chaineenliste) #\E) ; il faut avoir défini chaîneenliste (cf supra)
(eqv? (cadr chaineenliste) #\e)
```

Pour connaître toutes les fonctions sur : Number, String, Character, ou Vector
voir l'Aide de DrRacket (Dans Reference Racket => Datatypes)

II Quelques fonctions supplémentaires à écrire et à tester…

a) sur la récursivité terminale et non terminale

1) Ecrire la fonction factoriel **fac** qui :
- a comme argument un entier positif : n
- calcule le produit 1 * 2 * 3 * ... + n = n

Exemple : `(fac 5) = 120` *qui correspond à 5 * 4 * 3 * 2 * 1 avec donc (fac 0) = 1*

2) Ecrire la fonction factoriel **fac_term** qui soit récursive terminale:
Qui sert de fonction lanceuse à la fonction qui fait le calcul : **fac_it**

3) Etudier le temps – donc avec la fonction time – de **fac** (récursive non terminale) puis de **fac_term** (récursive terminale) pour n= 10000, 20000, 30000 40000 etc.

b) toujours sur la programmation de factoriel

4) Avec la fonction **fac** (ou **fac_term**) faire calculer (fac 20000)
Le résultat est évidemment gigantesque mais quel est son ordre de grandeur ?
Ecrire la fonction **ordre_de_grandeur** qui :
- a pour argument un entier positif : n
- retourne l'ordre de grandeur de n (rappel l'ordre de grandeur est la puissance de 10)

Exemple
(ordre_de_grandeur 1515) = 3 *(ordre_de_grandeur (fac_term 5)) = 2*
*Remarque : Il est possible de programmer **ordre_de grandeur** en une seule ligne de programmation !*
*Pour vérifier les résultats, on pourra utiliser la fonction prédéfinie de DrRacket : **order-of-magnitude***

c) A propos de réels en DrRacket

5) un nombre réel en DrRacket s'écrit sous la forme scientifique *mantisse* e *exposant.*
Exemples `(define Avogadro 6.02e23)` `(define chargeelem 1.6e-19)`
Mais en machine les réels sont limités à une valeur maximale et au delà l'ordinateur affiche : +inf.0

Par exemple, sachant que la fonction exact->inexact transforme un entier en un réel :

```
> (fac_term 100)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
> (exact->inexact (fac_term 100))
9.332621544394415e+157
> (exact->inexact (fac_term 1000))
 +inf.0
; par contre DrRacket est capable de calculer (fac_term 1000) – et même pour n beaucoup plus grand -  comme un entier !
> (= (exact->inexact (fac_term 1000)) (exact->inexact (fac_term 2000)))
#t
```

Ecrire la fonction **max_ordre_reel** qui :
- n'a aucun argument
- mais retourne l'ordre de grandeur **du plus grand réel** accessible en machine

Pour des réels à puissance négative, plus la puissance négative est grande en norme, plus le nombre est petit : pour une certaine puissance, le nombre est tellement petit qu'il est considéré par **DrRacket** comme égal à 0.

Ecrire la fonction **min_ordre_reel** qui :
- n'a aucun argument
- mais retourne l'ordre de grandeur **du plus petit réel** accessible en machine

c**) manipulation de listes**

On supposera pour la suite des fonctions à écrire que les listes sont des listes d'entiers relatifs et pour chaque fonction ci-dessous à écrire, on écrira une version qui fonctionne sur une liste plate, puis une deuxième version qui fonctionne sur une liste non plate -sauf 6) -

6) Ecrire les fonctions utiles (pour la suite)
   a) **cree_liste_integ** ayant un argument *n* : qui cree la liste de tous les entiers de 1 à n
   b) **cree_liste_random** ayant deux arguments *n* et *n_max* : qui cree une liste de n entiers ayat une valeur aléatoire dans l'intervalle [0, n_max[

Remarque : il existe une fonction prédéfinie (random N) qui retourne un nombre entier aléatoire dans l'intervalle [0, N[

7) Ecrire la fonction **occurrence** (puis **occurrence_gene sur liste non plate)** qui :
   - a pour argument un nombre *nb* et une liste *liste*
   - retourne le nombre d'occurrence de *nb* dans *liste*

8) Ecrire la fonction **enleve** (puis **enleve_gene sur liste non plate)** qui :
   - a pour argument un nombre *nb* et une liste *liste*
   - retourne la liste dans laquelle on a enlevé toutes les occurrences de *nb* dans *liste*

9) Ecrire la fonction **remplace** (puis **remplace_gene sur liste non plate)** qui :
   - a pour argument deux nombres *nb_a_rem et nb_rem* et une liste *liste*
   - retourne la liste dans laquelle on a remplacé toutes les occurrences de *nb* dans *liste*

10) Ecrire la fonction **elimine_doublon** qui :
   - a pour argument une liste *liste*
   - retourne la liste dans laquelle on a éliminé toutes les occurrences multiples dans *liste*

(Ecrire **elimine_doublon _gene** sur liste non plate en dernier.. car c'est la fonction la plus difficile du TD n°8**)**

11) Ecrire la fonction **remplace_lis** (puis **remplace_lis sur liste non plate)** qui :
   - a pour argument trois listes *liste_nb_a_rem, liste_nb remp* et *liste*
   - retourne la liste dans laquelle on a éliminé toutes les occurrences des nombres de la *liste_nb_a_rem* par tous ceux de la *liste_nb remp* dans *liste* (il est imposé que les deux listes liste_nb_a_rem, liste_nb remp aient la même longueur pour pouvoir opérer les remplacements en correspondance)

12) Ecrire la fonction **filtre** (uniquement sur liste **plate)** qui :
   - a pour argument une *liste et proc? :*une fonction à résultat booléen
   - retourne la liste issue de *liste* dans laquelle ne sont retenus que les éléments pour lesquels le prédicat *proc?* est vrai