LI101 Programmation récursive
Recueil d'exercices

Année universitaire 2011/2012

Table des matières

Chapitre 1

Exercices pour le premier TME

Premiers pas en Scheme

Exercice 1.

Question 1

1. Allez sur le site web [1] annuel de l'UE LI101 dont l'adresse est
   `http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2012/ue/LI101-2012oct/`
2. Ajoutez cette page à vos favoris, vous aurez à y venir souvent !
3. Sur le site, un lien vers l'application `MrScheme` est disponible : cliquez dessus !
4. L'icône avec un ? vous permet d'accéder à la documentation de `MrScheme`.
5. Sous les icônes, se trouve la fenêtre dite de définitions, c'est dans cette fenêtre que vous devrez programmer ou entrer des expressions Scheme.

Question 2

1. Tapez dans la fenêtre de définitions de MrScheme les *applications* [2] suivants :

```
"Tests d'égalité"
(equal?  "essai" "essai")
(equal? "essai" "essai ")
(equal? "3" 3)
(equal? 4 5)
(equal? 4 4)
(equal? 4 (* 2 2))

"Tests d'égalité numérique"
(= "essai" "essai")
(= 4 5)
(= 4 4)
(= (+ 2 2) 4)
```

1. Attention, il faut pour cela configurer proprement votre navigateur (les informations sont affichées en salles de tme).
2. la signification des mots en italiques dans ce texte de TME doit vraiment être comprise (voir le cours et la carte de référence)

```
"Tests sur number?"
(number? 45.12)
(number? -45)
(number? (+ 40 5))
(number? "essai")
(number? "3")

"Tests sur positive?"
(positive? 45.12)
(positive? -45)
(positive? (+ 40 5))
(positive? "essai")

"Tests sur remainder"
(remainder 10 5)
(remainder 5 10)
(remainder 10 0)

"Tests d'applications"
(= 3)
(3 + 2)
(3)
3
```

2. Sauvegardez la fenêtre de définitions dans un fichier intitulé « tme1.scm ». Pour cela, cliquez sur `Exporter le programme` (3ème icone au dessus de la fenêtre de définitions), ce qui a pour effet d'ouvrir un nouvel *onglet* dans lequel les définitions apparaissent [3]. Sauvegardez ensuite cet onglet sous forme de fichier à l'aide du menu déroulant en lui donnant le nom « tme1.scm ». Tous vos fichiers contenant des définitions ou applications *Scheme* doivent avoir un nom se terminant par « .scm ».
3. Cliquez sur le bouton `Exécuter le programme` (icone en haut à droite de la fenêtre de définitions) : les expressions de la fenêtre de définitions sont évaluées.
4. Observez le résultat des *évaluations* qui s'affichent en dessous de la fenêtre de définitions. Où s'est arrêtée l'évaluation ?
5. Regardez de nouveau dans la fenêtre de définitions : l'expression qui a provoqué le message d'erreur est surlignée en rouge ; aidez-vous des *spécifications* des fonctions données dans la carte de référence pour comprendre le message d'erreur.
6. Mettez un point virgule de commentaire devant l'expression ayant entraîné l'erreur avec une explication courte mais explicite sur l'origine de l'erreur (par exemple erreur de type, division par zéro...).
7. Sauvegardez.
8. Pour chacune des expressions évaluées correctement (avant l'expression incorrecte traitée juste avant), regardez dans la carte de référence la spécification des fonctions utilisées, auriez vous prédit/comprenez vous le résultat de l'évaluation à l'aide de l'expression et des spécification ? Si non pourquoi ?
9. Cliquez de nouveau sur le bouton `Exécuter le programme` (icone en haut à droite de la fenêtre de définitions)
10. Recommencez l'analyse des expressions correctes évaluées et le traitement des expressions qui provoquent un message d'erreur jusqu'à ce que toutes les lignes de la fenêtre de définitions soient évaluées par MrScheme (ou ignorées si elles ont été mises en commentaire).

3. cette procédure correspond à l'utilisation du navigatuer *firefox*, elle peut être légèrement différente avec d'autres navigateurs.

Remarque : prenez l'habitude de TOUJOURS sauvegarder la fenêtre de définitions avant d'exécuter. Lors de chaque séance de TME, prenez aussi l'habitude de nommer (typiquement « tmeX.scm » pour la Xième séance de TME) le fichier dans lequel vous allez programmer vos définitions ou écrire des applications de fonction dès le début de la séance et de le sauvegarder régulièrement.

Question 3

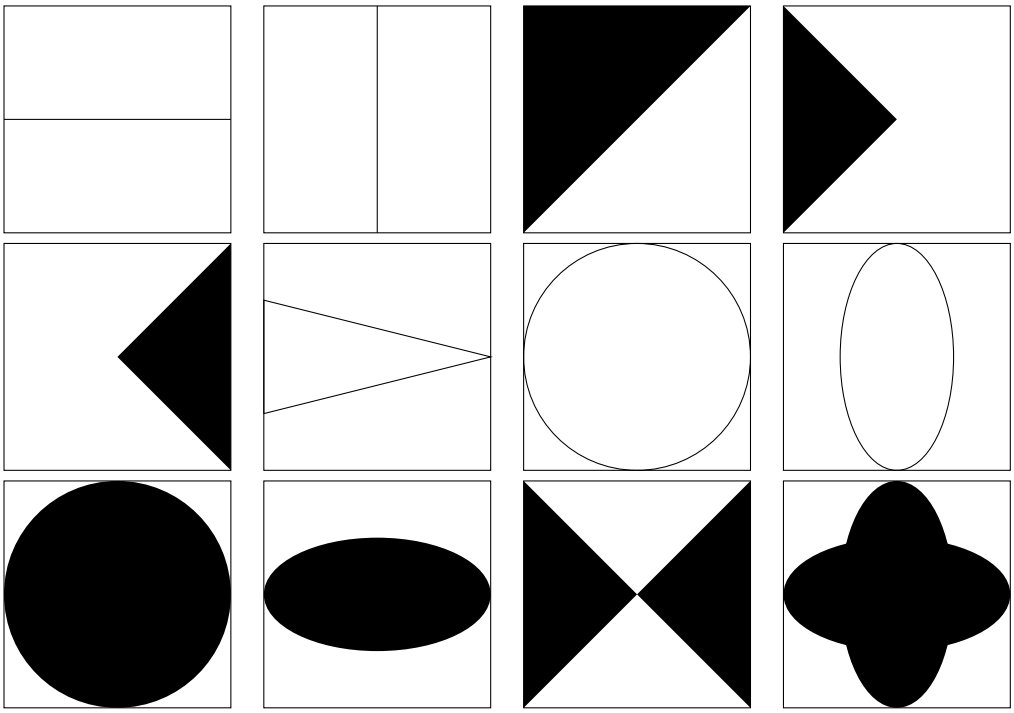
1. En mettant à chaque fois en commentaire le *nom de la fonction* utilisée et *l'argument* utilisé, écrivez (en vous aidant de la carte de référence) :
   - une application qui teste si le nombre 45.12 est un entier ;
   - une application correspondant à la négation de #t ;
   - une application qui renvoie le maximum de deux nombres ;
   - une application qui renvoie le quotient de la division euclidienne (entière) de 10 par 3 ;
   - une application qui renvoie le reste de la division euclidienne (entière) de 20 par 7 ;
2. Écrivez :
   - plusieurs *expressions* avec la fonction « `>` » ;
   - plusieurs expressions avec les fonctions `odd?` et `even?` ;
   - plusieurs expressions avec la fonction « `+` » en variant le nombre d'arguments, par exemple avec 1, 2, 3, 4 arguments et aussi avec 0 argument !
3. Entrez des expressions `Scheme` correspondant aux expressions ci-dessous et puis faites-les évaluer par MrScheme.
   - 10*10*10+9*10*10+3*10+6
   - ((1*10+9)*10+3)*10+6
4. Rappel : sauvegardez la fenêtre de définitions.

Question 4

Dans les questions qui suivent vous devez utiliser les fonctions graphiques décrites dans la carte de référence. Avant tout test, lisez bien les spécifications des différentes fonctions afin de savoir quelles sont les fonctions à votre disposition. Vous devez écrire des applications de fonctions graphiques en tentant de minimiser le nombre d'appels aux fonctions (on peut superposer deux triangles pour en former un troisième mais autant dessiner le triangle plein directement !).

Remarque : les coordonnées d'un point dans une image sont des nombres qui doivent appartenir à l'intervalle $[-1, +1]$.

1. Écrivez des applications de fonctions graphiques pour obtenir les dessins suivants dans la fenêtre d'interactions.



2. N'oubliez pas de sauvegarder la fenêtre de définitions.

Question 5

Cet exercice a pour but de montrer combien il est important de préciser la signification des arguments dans la spécification d'une fonction.

Soient les *spécifications* et *définitions* de deux fonctions `fill-rectangle1` et `fill-rectangle2` suivies de quatre *applications* de ces fonctions :

```
;;;  fill-rectangle1 : Nombre * Nombre * Nombre * Nombre -> Image
;;;  (fill-rectangle1 x y z t) produit une image carrée blanche contenant
;;;  un rectangle noir
```

```
(define (fill-rectangle1 x y z t)
  (overlay
   (fill-triangle x y z t x t)
   (fill-triangle x y z t z y) ) )
```

```
;;; fill-rectangle2 : Nombre * Nombre * Nombre * Nombre -> Image
;;; (fill-rectangle2 x y z t) produit une image carrée blanche contenant
;;; un rectangle noir.
```

```
(define (fill-rectangle2 x y z t)
  (overlay
   (fill-triangle x y (+ x z) (+ y t) x (+ y t))
   (fill-triangle x y (+ x z) (+ y t) (+ x z) y) ) )
```

```
(fill-rectangle1 0 0 0.45 0.75)
(fill-rectangle2 0 0 0.45 0.75)
(fill-rectangle1 -0.45 -0.45 0.60 0.45)
(fill-rectangle2 -0.45 -0.45 0.60 0.45)
```

1. Quelles images obtient-on dans la fenêtre d'interactions ?
2. Complétez la spécification de chacune de ces fonctions pour préciser la signification de leurs arguments. Toutes les valeurs de paramètres sont elles possibles ? Corrigez si besoin votre spécification.

Chapitre 2

Exercices simples

Exercice 2. Soustraire 1

En route pour le premier exercice...

Question 1

Cette définition est-elle syntaxiquement correcte ?

```
(define (enleve-un n)
  (- 1 n))
```

La spécification suivante correspond-elle à la fonction `enleve-un` définie ci-dessus ?

```
;;; enleve-un : Nombre -> Nombre
;;; (enleve-un n) rend n-1
```

Corriger le corps de la fonction `enleve-un` pour que sa définition corresponde bien à cette spécification.

Exercice 3. Période d'un pendule

Cet exercice a pour but d'écrire la *spécification* et une *définition* de la fonction qui calcule la période d'un pendule.

Question 1

La période p (en secondes) d'un pendule de longueur L (en cm), s'obtient en multipliant par 2π la racine carrée du quotient L/g, où g est la constante de gravitation, égale à 981 cm/s^2.

Écrire la *spécification* et une *définition* Scheme de la fonction `periode` qui, étant donnée une longueur, retourne la période d'un pendule de cette longueur.

Écrire aussi l'*application* (appel de fonction) pour calculer la période d'un pendule de 3 cm de longueur ; même question pour des longueurs de 4 cm et 5,2 cm.

Remarque : dans DrScheme, le nombre π est déjà défini ; il s'appelle `pi` et vaut 3.141592653589793.

Exercice 4. Conversion degrés Fahrenheit-Celsius

Cet exercice a pour but d'écrire la spécification, un jeu de tests et la définition de fonctions très simples.

Question 1

Écrire la *spécification*, un jeu de tests (c'est-à-dire un ensemble d'applications pertinentes) et une *définition* Scheme de la fonction, nommée `Fahrenheit->Celsius`, qui effectue la conversion des degrés Fahrenheit en degrés Celsius : lorsque la température est t en degrés Fahrenheit, elle est égale à $(t - 32) * 5/9$ en degrés Celsius.

Question 2

Écrire un jeu de tests et une *définition* Scheme de la fonction `Celsius->Fahrenheit` qui effectue la conversion en sens inverse.

Exercice 5. Aire d'une couronne

Cet exercice a pour but de calculer l'aire d'une couronne, et de travailler sur les notions d'hypothèse et d'erreur.

Question 1

Proposer la *spécification*, un jeu de tests et une *définition* Scheme de la fonction `aire-disque` telle que `(aire-disque r)` calcule l'aire du disque de rayon r.

Remarque : dans DrScheme, le nombre π est déjà défini ; il s'appelle `pi` et vaut 3.141592653589793.

Question 2

Proposer la *spécification*, un *jeu de tests* et une *définition* Scheme de la fonction `aire-couronne1` telle que `(aire-couronne1 r1 r2)` calcule l'aire d'une couronne de rayon intérieur `r1` et de rayon extérieur `r2`. On supposera ici que le rayon intérieur donné est toujours inférieur ou égal au rayon extérieur donné.

Question 3

Écrire la *spécification* et une *définition* de la fonction `aire-couronne2` qui, comme `aire-couronne1`, calcule l'aire d'une couronne de rayon intérieur `r1` et de rayon extérieur `r2`, mais retourne une erreur lorsque `r1 > r2`.

Exercice 6. Calcul d'un prix TTC

Cet exercice a pour but de trouver les paramètres d'un problème, et d'écrire la spécification, un jeu de tests et la définition de fonctions très simples.

Question 1

Écrire la *spécification*, un jeu de tests et une *définition* Scheme de la fonction `prix-ttc` qui calcule le prix toutes taxes comprises (TTC) à partir d'un prix hors taxe (HT) et d'un taux exprimé en pourcentage (par exemple 18 pour 18%).

Question 2

Écrire la *spécification*, un jeu de tests et une *définition* Scheme de la fonction inverse `prix-ht` qui calcule le prix hors taxe étant donnés le prix toute taxe comprise et le taux.

Exercice 7. Moyenne de trois nombres

Cet exercice a pour but de trouver les paramètres d'une fonction pour résoudre un problème, et d'écrire la spécification et la définition de fonctions très simples.

Question 1

Donner la *spécification* et une *définition* `Scheme` de la fonction `moyenne` qui effectue la moyenne arithmétique de trois nombres.

Écrire aussi les *applications* (appels de fonction) pour calculer la moyenne de 3, 6 et -3 puis de -3, 0 et 3 puis de 1, 2 et 1.0.

Question 2

Écrire la *spécification* et une *définition* Scheme de la fonction `moyenne-ponderee` qui effectue la moyenne de trois nombres avec des coefficients variables.

Donner aussi des *applications* utilisant cette fonction.

Exercice 8. Quotient et reste

Petit rappel d'arithmétique : division *eucidienne* ou entière, quotient et reste.

En arithmétique, pour tout entier n et m, si $m \neq 0$, il existe q et r tels que $n = qm + r$. On appelle q le *quotient* de la division euclidienne (ou entière) de n par m et r le *reste*.

En Scheme, on a les fonctions :

```
;;; quotient : int * int -> int
;;; (quotient a b) rend le quotient de la division euclidienne de a par b.
;;; ERREUR lorsque b est égal à 0.

;;; remainder : int * int -> int
;;; (remainder a b) rend le reste de la division euclidienne de a par b.
;;; ERREUR lorsque b est égal à 0.
```

Question 1

En utilisant les fonctions `quotient` et `remainder` ainsi que deux variables `n` et `m`, écrire une expression Scheme pour l'égalité arithmétique : $n = qm + r$.

Question 2

Donnez et expliquez les résultats des *applications* suivantes :

```
(quotient 15 2)
(remainder 15 2)
(quotient 15 3)
(remainder 15 3)
(quotient 15 4)
(remainder 15 4)
(quotient 15 27)
(remainder 15 27)
```

Question 3

Quotient et reste sont aussi définis pour les nombres négatifs. Donnez et expliquez les résultats des application suivantes :

```
(quotient 15 -4)
(remainder 15 -4)
(quotient -15 4)
(remainder -15 4)
(quotient -15 -4)
(remainder -15 -4)
```

Question 4

Division euclidienne par 10 et *numération décimale* (écriture des chiffres en *base 10*).
On a :

$$\begin{aligned} 456 &= 4 \times 10 \times 10 + 5 \times 10 + 6 \\ &\quad (4 \times 10 + 5) \times 10 + 6 \end{aligned}$$

- En utilisant `remainder` et `quotient`, écrire trois expressions Scheme telles que la première donne le chiffre des unités, la deuxième donne le chiffre des dizaines et la troisième le chiffre des centaines du nombre 456.
- Même question pour les chiffres du nombre 53980.

Exercice 9. Sur l'alternative

Cet exercice vous fera travailler l'alternative c'est-à-dire la forme *si...alors...sinon*.

Question 1

Soit la définition suivante de la fonction `f` :

```
(define (f x y z)
  (if (> x y)
      (if (> x z)
          x
          z)
      (if (> z y)
          z
          y)))
```

En étudiant la définition donnée ci-dessus, déduire le nombre et le type des paramètres de la fonction ainsi que ce que calcule la fonction. À partir de cette étude, renommer la fonction `f` avec un nom plus parlant et donner sa spécification complète.

Question 2

Chercher dans la carte de référence une fonction sur les nombres permettant de calculer la même chose et utiliser la pour définir une fonction `f-bis` de même spécification que `f`.

Question 3

Donner une définition Scheme de la fonction `f-ter` de même spécification que `f` mais qui utilise la relation `<=` au lieu de `>`.

Exercice 10. Calcul des mentions

Cet exercice a pour but de faire manipuler des *alternatives* imbriquées et la forme spéciale conditionnelle `cond`. Il s'agit de calculer la mention correspondant à une note sur 20.

Question 1

Donner la spécification et une définition Scheme de la fonction `mention` qui calcule la mention correspondant à une note (sur 20) donnée. Vous utiliserez des alternatives imbriquées pour cette question. Par exemple :
`(mention 8.5)` → `"Eliminé"`
`(mention 10)` → `"Passable"`
`(mention 12.5)` → `"AB"`
`(mention 15)` → `"B"`

`(mention 16.5)` → `"TB"`
Les mentions sont attribuées selon les intervalles de notes suivants :

$[0, 10[$	`"Eliminé"`
$[10, 12[$	`"Passable"`
$[12, 14[$	`"AB"`
$[14, 16[$	`"B"`
$[16, 20]$	`"TB"`

Question 2

Écrire une autre définition de la fonction `mention` (avec la même spécification), qui utilise la conditionnelle `cond`.

Exercice 11. Manipulation des booléens

Cet exercice a pour but de définir des fonctions sur les booléens en utilisant des constructions alternatives.

Question 1

Sans utiliser les formes spéciales and et or, ni la fonction not, écrire les trois fonctions booléennes `ou`, `et` et `non` de spécifications :

```
;;; ou : bool * bool -> bool
;;; (ou x y) rend true lorsque x ou y vaut true

;;; et : bool * bool -> bool
;;; (et  x y) rend true lorsque x et y valent true

;;; non : bool -> bool
;;; (non x) rend true ssi x vaut false
```

Question 2

En utilisant les fonctions précédentes, écrire les fonctions booléennes `implique` et `ou-exclusif` de spécifications :

```
;;; implique : bool * bool -> bool
;;; (implique a b) rend true lorsque a vaut false ou lorsque b vaut true

;;; ou-exclusif : bool * bool -> bool
;;; (ou-exclusif a b) rend true ssi une et une seule des valeurs a et b  vaut true
```

Question 3

Écrire la fonction booléenne `equivalent` de spécification :

```
;;;  equivalent : bool * bool -> bool
;;;  (equivalent a b) rend true ssi a et b ont la même valeur
```

Exercice 12. Petit algorithme de vote

Question 1

Donnez la spécification et la définition de la fonction `abs` qui donne la valeur absolue de son argument.

Question 2

On cherche à savoir si deux valeurs $x1$ et $x2$ sont égales *à epsilon près*, c'est-à-dire si la valeur absolue de leur différence est inférieure à une valeur positive donnée (supposée petite).

1. Donnez la spécification et la définition du prédicat `eps-equal?` qui teste l'égalité de deux valeurs *à epsilon près*. La valeur de cet *epsilon* peut être passée en argument.
2. Donnez un ensemble de tests le plus complet possible pour votre fonction. Vous devez tenir compte du maximum de cas de figure possibles : arguments positifs ou négatifs, rapport entre les deux arguments (plus petit ou plus grand), valeur attendue vraie ou fausse.

Question 3

Pour fiabiliser une mesure, il est bon de ne pas se fier à une seule mesure. On suppose donc que l'on dispose de trois capteurs fournissant trois valeurs numériques $v1$, $v2$ et $v3$ censées donner chacune la mesure d'un même phénomène (une température par exemple). On tolère une différence jugée négligeable entre ces valeurs : en d'autres termes, on considère leur égalité *à epsilon près*.

Attention : l'égalité *à epsilon près* n'est pas transitive en général, elle reste en revanche symétrique.

On cherche à déterminer un taux de fiabilité de la mesure en appliquant le principe suivant :

- si les trois valeurs sont deux à deux égales (à epsilon près), le taux est de $3/3$;
- si deux couples de valeurs seulement sont égales (à epsilon près), le taux de fiabilité est de $2/3$ (par exemple : $v1 \approx v3$ et $v2 \approx v3$) ;
- sinon, le taux de fiabilité est nul (0).

1. Combien y-a-t'il de façons d'obtenir un taux de fiabilité de $3/3$, de $2/3$ et de 0 ? Donnez des exemples.
2. Donnez la signature, ainsi que l'hypothèse, et une définition de la fonction `fiabilite` qui donne le taux de fiabilité des trois valeurs à un epsilon près.

Exercice 13. Couverture

On cherche dans cet exercice à définir un jeu de tests permettant d'explorer tous les cas de tests d'une fonction.

Question 1

La fonction suivante rend 6 valeurs différentes (une chaîne de caractères) selon l'ordre dans lequel sont donnés ses trois arguments.

```
(define (f n1 n2 n3)
  (if (and (< n1 n2) (< n2 n3)) "cas 1"
      (if (and (< n1 n3) (< n3 n2)) "cas 2"
          (if (and (< n2 n1) (< n1 n3)) "cas 3"
              (if (and (< n2 n3) (< n3 n1)) "cas 4"
                  (if (and (< n3 n1) (< n1 n2))
                      "cas 5"
                      "cas 6"))))))
```

Donnez six exemples d'application de `f` permettant d'obtenir les six chaînes de caractères.

Question 2

Donnez une autre définition de `f` sans utiliser la forme spéciale `and` (ni `or`). Vérifiez que vous obtenez bien les mêmes résultats avec le jeu de tests de la question précédente.

Exercice 14. Prédicat pour un nombre positif

Cet exercice a pour but d'écrire un *prédicat* qui indique si un nombre est positif ou non.

Question 1

Écrire la *signature* et une *définition* Scheme du prédicat `positif?` qui, étant donné un nombre n retourne vrai si, et seulement si, n est strictement positif. Par exemple :

```
(positif? 3) → #t
(positif? -3) → #f
(positif? 0) → #f
```

Question 2

Écrire la *signature* et une *définition* Scheme du prédicat `nombre-positif?` qui, étant donnée une *valeur* quelconque x, retourne vrai si, et seulement si, x est un nombre et s'il est strictement positif. Par exemple :

```
(nombre-positif? 3) → #t
(nombre-positif? -3) → #f
(nombre-positif? "toto") → #f
```

Exercice 15. Prédicats pour division entière

Cet exercice a pour but d'écrire des prédicats et semi-prédicats vérifiant qu'un nombre divise un autre nombre.

Question 1

En n'utilisant pas les prédicats prédéfinis `even?` et `odd?`, écrire la *spécification* et une *définition* Scheme du prédicat `nombre-pair?` qui, étant donné un nombre entier, rend vrai si, et seulement si, ce nombre est pair. Par exemple :

```
(nombre-pair? -10) → #t
(nombre-pair? 0) → #t
(nombre-pair? 6) → #t
(nombre-pair? 5) → #f
```

Question 2

Écrire la *spécification* et une *définition* Scheme de la fonction `divise?` qui, étant donnés un entier strictement positif m et un entier n rend vrai si, et seulement si, m divise n. Par exemple :

```
(divise? 3 12) → #t
(divise? 6 35) → #f
(divise? 8 2) → #f
```

Question 3

Écrire une autre définition de la fonction `nombre-pair?`, qui utilise la fonction `divise?`.

Question 4

Un *semi-prédicat* est une fonction qui renvoie soit `#f`, soit la valeur vrai au travers d'une valeur quelconque différente de `#f`.

Écrire la *spécification* et une *définition* Scheme du *semi-prédicat* `quotient-si-divise` qui, étant donnés un entier strictement positif m et un entier n rend le quotient de n par m si m divise n et le booléen faux sinon. Par exemple :

```
(quotient-si-divise 3 12) → 4
```

`(quotient-si-divise 6 35)` → `#f`
`(quotient-si-divise 8 2)` → `#f`
`(quotient-si-divise 2 -10)` → `-5`

Question 5

Écrire une *définition* Scheme du semi-prédicat `produit-diviseur-quotient-si` qui, étant donnés un entier strictement positif m et un entier n rend le produit de m et du quotient de n par m si m divise n et le booléen faux sinon. Par exemple :
`(produit-diviseur-quotient-si 3 12)` → `12`
`(produit-diviseur-quotient-si 6 35)` → `#f`
`(produit-diviseur-quotient-si 8 2)` → `#f`
`(produit-diviseur-quotient-si 2 -10)` → `-10`

Exercice 16. Calcul de fonctions polynômiales

Cet exercice a pour but de *définir* des fonctions simples de calcul de polynômes. On mettra l'accent sur l'*efficacité* (minimisation du nombre de multiplications).

Question 1

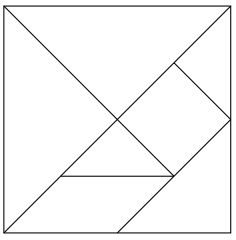
Après avoir spécifié le problème, écrire un jeu de tests et une *définition* `Scheme` de la fonction, nommée `polynomiale`, telle que `(polynomiale a b c d x)` rend la valeur de la fonction qui à x associe $ax^3 + bx^2 + cx + d$. Par exemple :
`(polynomiale 1 1 1 1 2)` → `15`
`(polynomiale 1 1 1 1 3)` → `40`
Quel est le nombre de multiplications effectuées par cette fonction ?

Question 2

Après avoir spécifié le problème, écrire un jeu de tests et une définition de `polynomiale-carre`, fonction qui rend la valeur de la fonction $ax^4 + bx^2 + c$. Par exemple :
`(polynomiale-carre 1 1 1 2)` → `21`
`(polynomiale-carre 1 1 1 3)` → `91`
Quel est le nombre de multiplications effectuées ?

Exercice 17. Tracer un tangram

Les pièces du jeu de *tangram* sont obtenues en divisant un carré de la façon suivante :



Question 1

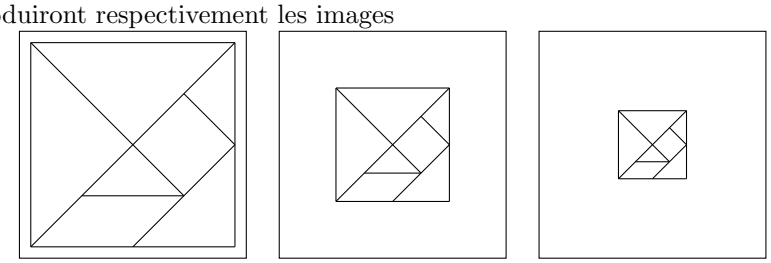
Donner la *spécification* et une *définition* Scheme de la fonction *sans argument* `tangram` qui trace le dessin des pièces du jeu de tangram.

Question 2

L'on veut pouvoir appliquer un facteur d'échelle au dessin du tangram. Il faut pour cela définir une nouvelle fonction `scale-tangram` prenant en paramètre le facteur d'échelle. Ainsi les applications

```
(scale-tangram 0.9)
(scale-tangram 0.5)
(scale-tangram 0.3)
```

produiront respectivement les images



Une manière simple de répondre à cette question est de reprendre le code de la définition de `tangram` en remplaçant l'utilisation de `draw-line` par une fonction `scale-draw-line` qui trace un segment sur lequel on applique un facteur d'échelle. Voici cette fonction :

```
;;; scale-draw-line : Nombre * Nombre^4 -> Image
;;; (scale-draw-line s x1 y1 x2 y2) rend l'image d'un segment
;;; d'extrémités s*x1, s*y1 et s*x2, s*y2.
(define (scale-draw-line s x1 y1 x2 y2)
  (draw-line (* x1 s) (* y1 s) (* x2 s) (* y2 s)))
```

De plus, la fonction `tangram` ne traçait pas les contours du tangram. Elle utilisait le contour par défaut de l'image. Il faut donc définir une fonction `scale-square` qui trace un carré (centré en $(0, 0)$) selon un facteur d'échelle.

1. donnez la *spécification* et une *définition* de la fonction `scale-square` ;
2. donnez la *spécification* et une *définition* de la fonction `scale-divide` qui trace les lignes de division du tangram en tenant compte d'un facteur d'échelle ;
3. donnez la *spécification* et une *définition* de la fonction `scale-tangram` qui rend l'image d'un tangram selon un facteur d'échelle donné en paramètre.

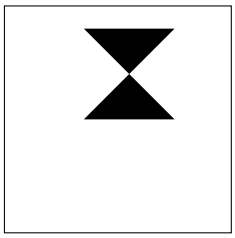
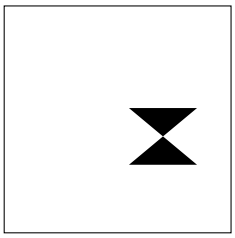
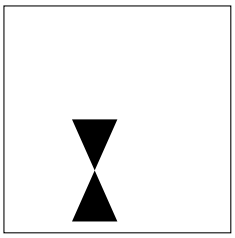
Exercice 18. Dessin d'un sablier

Le problème à résoudre dans cet exercice est de dessiner un sablier.

L'objectif de cet exercice est de travailler sur les *paramètres* nécessaires pour la définition d'une fonction permettant de résoudre un problème : il faut, bien sûr, qu'il y ait tous les paramètres nécessaires à la résolution du problème, mais il ne faut pas qu'il y en ait en plus. Autrement dit les différents paramètres doivent être **indépendants** et l'on doit donc pouvoir appeler la fonction avec n'importe quelles valeurs respectant les hypothèses pour que le sablier ne dépasse pas les bornes (ici, les coordonnées sont comprises entre -1 et 1).

Question 1

Nous voudrions définir une fonction qui retourne l'image d'un sablier. Nous voulons que cette fonction permette d'obtenir des sabliers de tailles et de positions différentes. Exemples :

 ou  ou 

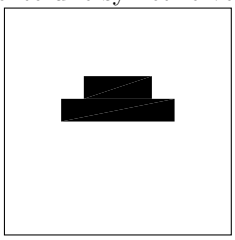
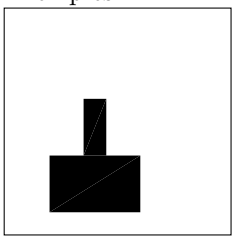
1. Quels sont les paramètres nécessaires ? En déduire une *spécification* de la fonction.
2. Donner un jeu de tests.
3. Donner une *définition* Scheme de la fonction correspondant à la spécification choisie.

Exercice 19. Dessin d'une tour

L'objectif de cet exercice est de travailler sur les paramètres nécessaires à un problème : bien sûr, il faut qu'il y ait tous les paramètres nécessaires à la définition du problème, mais il ne faut pas qu'il y en ait en plus. Autrement dit les différents paramètres doivent être indépendants. Encore autrement dit, on doit pouvoir appeler une telle fonction avec n'importe quelles valeurs, sous réserve qu'elles ne dépassent pas les bornes (ici, les coordonnées sont comprises entre -1 et 1).

Question 1

Nous voudrions définir une fonction qui retourne l'image de tours à deux étages de tailles et positions différentes. Les deux étages d'une même tour ont la même hauteur et chaque tour présente une symétrie verticale. Exemples :

 ou 

1. Quels sont les paramètres nécessaires ?
2. Donner une définition Scheme de la fonction.

Chapitre 3

Récursion sur les entiers

Exercice 20. Quelle est cette fonction f ?

Une récursion simple sur les entiers.

Question 1

Soit la fonction `f`, partiellement définie par :

```
(define (f n)
  (if (= n 0)
      ; cas de base
      ; cas général
  )
)
```

1. Compléter cette définition pour que
   - dans le *cas de base* elle renvoie 0,
   - dans le *cas général* elle renvoie la somme de n^2 et du résultat de l'appel récursif de f sur $n-1$.
2. Faire *tourner à la main* l'évaluation de l'application `(f 4)`.
3. Faire *tourner à la main* l'évaluation de l'application `(f -1)`.
4. En déduire la spécification de cette fonction que vous renommerez avec un nom plus approprié.

Exercice 21. Quelle est cette fonction g ?

Une récursion simple sur les entiers : fonction avec deux paramètres.

Question 1

Soit la fonction `g`, partiellement définie par :

```
(define (g m n)
  (if ; condition
      0
      (+ (* m m) (g (+ m 1) n))))
```

1. Compléter cette définition pour que la *condition* rende vrai lorsque m est strictement supérieur à n.
2. Faire *tourner à la main* l'évaluation de l'application `(g 3 5)`.

3. Faire *tourner à la main* l'évaluation de l'application `(g 5 3)`.
4. En déduire la spécification complète de cette fonction que vous renommerez avec un nom plus parlant.

Exercice 22. Somme des n premiers impairs

Question 1

Écrire la *spécification* et une *définition récursive* de la fonction `somme-impairs` qui, étant donné un entier strictement positif n, rend la somme des n premiers entiers impairs : $1+3+5+\ldots+2n-1$. Par exemple :

`(somme-impairs 1)` → `1`
`(somme-impairs 7)` → `49`

Question 2

Écrire la *signature* et une *définition* d'un prédicat `verif-formule?` qui, étant donné un entier positif n, vérifie que $1 + 3 + 5 + \ldots + 2n - 1 = n^2$.

Exercice 23. Nombres et chiffres

Cet exercice demande la définition de fonctions récursives sur les nombres et leur notation décimale ou binaire. On mettra l'accent sur les conditions d'arrêt, ou de poursuite, des appels récursifs.

Question 1

Écrire la *spécification* et une *définition* de la fonction `somme-des-chiffres` qui rend la somme des chiffres d'un entier naturel n. Par exemple :

`(somme-des-chiffres 546)` → `15`
`(somme-des-chiffres 17)` → `8`
`(somme-des-chiffres 0)` → `0`

Question 2

Écrire la *spécification* et une *définition* de la fonction `nombre-de-chiffres` qui rend le nombre de chiffres significatifs d'un entier naturel. Par exemple :

`(nombre-de-chiffres 546)` → `3`
`(nombre-de-chiffres 7)` → `1`
`(nombre-de-chiffres 0)` → `1`

Question 3

Écrire la *spécification* et une *définition* du prédicat `existe-chiffre?` qui, étant donné un chiffre c entre 0 et 9, et un entier naturel, rend vrai si et seulement si le chiffre c apparaît dans l'écriture en base 10 de n. Par exemple :

`(existe-chiffre? 5 546)` → `#t`
`(existe-chiffre? 0 546)` → `#f`
`(existe-chiffre? 0 6045)` → `#t`

Question 4

Écrire la *spécification* et une *définition* de la fonction `nombre-de-bits` qui rend le nombre de bits significatifs (chiffre binaire : 0 ou 1) dans l'écriture en base 2 d'un entier naturel n donné en base 10. Par exemple :

`(nombre-de-bits 0)` → `1`

`(nombre-de-bits 2)` → `2`
`(nombre-de-bits 7)` → `3`
`(nombre-de-bits 32)` → `6`

Question 5

Écrire la *spécification* et une *définition* de la fonction `nombre-de-chiffres-dans-base` qui, étant donnés une base $B > 1$ et un entier naturel n, rend le nombre de "chiffres" significatifs nécessaires à l'écriture en base B de n (en base B il y a B chiffres possibles : $0, 1, \ldots, B-1$). Par exemple :
`(nombre-de-chiffres-dans-base 2 59)` → `6`
`(nombre-de-chiffres-dans-base 7 59)` → `3`
`(nombre-de-chiffres-dans-base 16 59)` → `2`

Exercice 24. Calcul du pgcd

L'objectif de cet exercice est d'écrire des fonctions calculant le *plus grand diviseur commun* (*pgcd*) de deux entiers (naturels) selon différents algorithmes.

Question 1

Un premier algorithme consiste à calculer le *pgcd* de deux entiers (naturels) par différences successives. On peut exprimer cet algorithme en utilisant l'ensemble d'équations suivantes :
- $pgcd(m, 0) = m$
- $pgcd(0, n) = n$
- $pgcd(m, n) = pgcd(m, n - m)$, si $m < n$
- $pgcd(m, n) = pgcd(m - n, n)$, sinon.

Donner la *spécification* et une *définition* de la fonction `pgcd` qui implante cet algorithme en utilisant uniquement des alternatives puis une version en utilisant la forme conditionnelle `cond`.

Question 2

Le deuxième algorithme proposé procède par divisions successives. Il utilise les égalités suivantes :
- $pgcd(m, 0) = m$
- $pgcd(m, n) = pgcd(n, m \ mod \ n)$, si $n \neq 0$

Donnez une *définition* de la fonction `pgcd-bis` qui implante l'algorithme défini par ces équations.

Question 3

On propose enfin d'implanter un algorithme de calcul du *pgcd* par *dichotomie* basé sur les équations suivantes :
- $pgcd(m, n) = 2 \times pgcd(m \div 2, n \div 2)$, si m et n sont pairs ;
- $pgcd(m, n) = pgcd(m \div 2, n)$, si m est pair et n impair.

Bien entendu, les équations
- $pgcd(m, 0) = m$
- $pgcd(0, n) = n$

restent valables.

On pourra traiter le cas où m est impair et n pair en remarquant que
- $pgcd(m, n) = pgcd(n, m)$

Enfin, pour le cas où ni m, ni n ne sont pairs, on utilisera le fait qu'alors la différence entre m et n est paire (pensez que dans ce cas, il faut soustraire le plus petit au plus grand).

Donnez une *définition* de la fonction `pgcd-ter` qui implante le calcul du *pgcd* par dichotomie.

Exercice 25. Suites récursives

Illustration financière du calcul des termes d'une suite définie par récurrence.

On dispose d'un capital initial K que l'on place à un taux d'intérêt t. On cherche à savoir quelle est la valeur du capital après n années.

Question 1

Donnez la formule de récurrence de la suite C_n pour $n \in \mathbb{N}$. La première année est l'année 0.

Question 2

Donnez la *spécification* et une *définition* de la fonction `valeur-capital` qui permet de calculer le n-ième élément de la suite définie ci-dessus.

Question 3

On suppose cette fois que l'on se fixe un versement annuel A à un taux t sur n années. On cherche à connaître le montant de l'épargne ainsi réalisée.

Donnez la formule de récurrence de la suite E_n, pour $n \in \mathbb{N}$, dont chaque terme E_n donne le montant de l'épargne obtenue à la i-ième année.

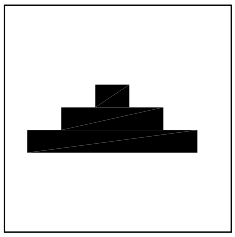
Question 4

Donnez la *spécification* et une *définition* de la fonction `valeur-epargne` qui permet de calculer le n-ième élément de la suite définie dans la question précédente.

Exercice 26. Pyramides

Le but de cet exercice est de dessiner des pyramides, avec un nombre quelconque d'étages. Lorsque nous dessinerons une pyramide, toutes ses marches auront la même largeur l et la même hauteur h.

Sur le dessin ci-contre, par exemple, on a dessiné une pyramide à trois étages.



Si une pyramide a n étages alors le dernier étage a la largeur d'une marche, l'avant-dernier a la largeur de trois marches, l'antépénultième a la largeur de cinq marches, ..., et le premier ? Si chaque marche a pour largeur l et pour hauteur h, quelle est la largeur totale de la pyramide ? sa hauteur totale ?

Indication : Le premier étage a la largeur de $2n - 1$ marches.
Un petit raisonnement par récurrence convaincra les hésitants :
– c'est vrai pour $n = 1$ (1 seul étage = 1 marche).
– si c'est vrai pour une pyramide à $n-1$ étages alors c'est vrai aussi pour une pyramide à n étages, puisque le premier étage de la pyramide à n étages compte, en largeur, 2 marches de plus que le premier étage de la pyramide à $n - 1$ étages.

La largeur totale de la pyramide est donc $(2n - 1)l$ et sa hauteur totale est nh.

Question 1

Écrire la *spécification* et une *définition* de la fonction `rectangle`, de paramètres x, y, l, h, qui rend l'image d'un rectangle de coin bas-gauche (x, y), de largeur l et de hauteur h.

Question 2

Définition récursive d'une pyramide.

Soit P une pyramide ayant n étages, de coin bas-gauche (x, y) et dont chaque marche a pour hauteur h et pour largeur l. La pyramide P se décompose en un rectangle R égal à l'étage du bas et une pyramide Q formée des $n - 1$ étages du haut. Quelles sont les dimensions du rectangle R ? les coordonnées de son coin bas-gauche ? Quelles sont les coordonnées du coin bas-gauche de la pyramide Q ?

Écrire la *spécification* et une *définition récursive* de la fonction `pyramide1`, de paramètres n, x, y, l, h, qui rend l'image d'une pyramide ayant n étages, de coin bas-gauche (x, y) et dont chaque marche a pour hauteur h et pour largeur l. Cette fonction ne signalera pas d'erreur.

Question 3

Écrire la *spécification* et une *définition* de la fonction `pyramide2`, de paramètres n, x, y, l, h, qui rend l'image d'une pyramide en vérifiant qu'elle tient dans le cadre de dessin de DrScheme, et signale une erreur si ce n'est pas le cas. Cette fonction indiquera la (ou les) dimension(s) trop grande(s) dans le message d'erreur.

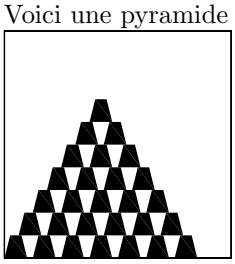
Question 4

Écrire la *spécification* et une *définition* de la fonction `pyramide3`, de paramètres n, x, y, l, h, qui rend l'image d'une pyramide *tenant toujours* à l'intérieur du cadre de dessin de DrScheme. Au besoin, cette fonction modifiera la taille des marches ou le nombre d'étages.

Exercice 27. Pyramides de gobelets

Un gobelet est représenté par un trapèze isocèle dont la grande base est égale à la hauteur et dont la petite base est égale à la moitié de la hauteur. Construire une pyramide avec des gobelets est un jeu d'enfant : on aligne une rangée de gobelets (renversés), puis on pose une rangée de gobelets à cheval sur les gobelets de la première rangée, et ainsi de suite. Si la pyramide a n étages, le dernier étage est formé d'un seul gobelet, l'avant-dernier étage est formé de deux gobelets, ..., le premier étage est formé de n gobelets.

Voici une pyramide de gobelets à sept étages :



Question 1

Écrire la *spécification* et une *définition* de la fonction `gobelet` de paramètres x, y, h, qui dessine un gobelet de coin bas-gauche (x, y) et de hauteur h. Les autres dimensions nécessaires pour dessiner un gobelet sont laissées au choix, mais doivent s'exprimer en fonction de h. Cette fonction ne signalera pas d'erreur.

Question 2

Écrire la *spécification* et une *définition* de la fonction `pyramide-gobelet1`, de paramètres n, x, y, h, e, qui dessine une pyramide de gobelets ayant n étages, de coin bas-gauche (x, y), h étant la hauteur d'un gobelet et e l'écart entre deux gobelets consécutifs d'un même étage. Cette fonction ne signalera pas d'erreur.

Question 3

Écrire la *spécification* et une *définition* de la fonction `pyramide-gobelet2`, de paramètres n, x, y, h, qui dessine une pyramide de gobelets ayant au plus n étages, de coin bas-gauche (x, y), h étant la hauteur d'un gobelet. La pyramide *doit tenir* à l'intérieur du cadre de dessin de DrScheme. La fonction `pyramide-gobelet2` utilisera la fonction `pyramide-gobelet1` : attention, l'écart entre deux gobelets consécutifs d'un même étage ne doit pas être trop grand (les gobelets ne tiendraient pas en équilibre) et le nombre d'étages ne doit pas lui non plus être trop grand (la pyramide *doit tenir* à l'intérieur du cadre de dessin de DrScheme).

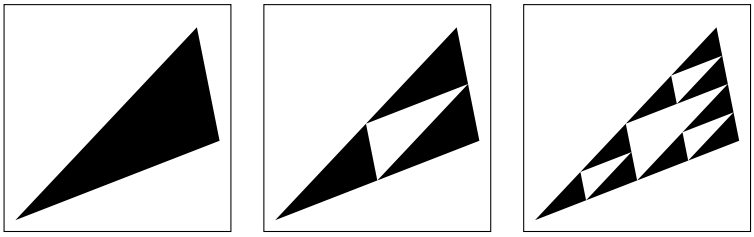
Question 4

Écrire la *spécification* et une *définition* de la fonction `pyramide-gobelet3`, de paramètres x, y, h, $nb - gob$, qui dessine une pyramide construite avec au plus $nb - gob$ gobelets, de coin bas-gauche (x, y), h étant la hauteur d'un gobelet. La pyramide *doit tenir* à l'intérieur du cadre de dessin de DrScheme.

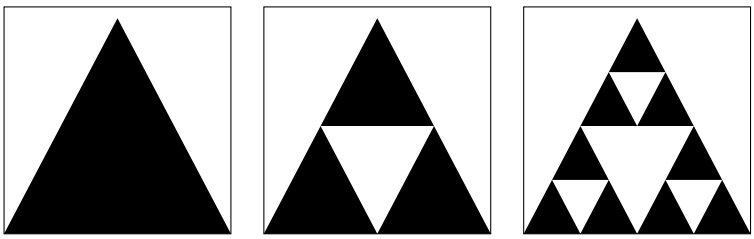
Exercice 28. Triangles de Sierpinski

Voici une définition récursive des *triangles de Sierpinski* :
- un triangle de Sierpinski de rang 0 est un triangle noir
- étant donnés trois points A, B, C (non alignés), on obtient un triangle de Sierpinski de rang $n + 1$ en construisant les trois triangles de Sierpinski $AB'C'$, $A'BC'$ et $A'B'C$, de rang n, où A', B' et C' sont les milieux respectifs des côtés $[BC]$, $[AC]$ et $[AB]$.

Voici trois triangles de Sierpinski de forme quelconque, un de rang 0, un de rang 1 et un de rang 2.



Et voici trois triangles de Sierpinski isocèles.



Question 1

Écrire la *spécification* et une *définition récursive* de la fonction `sierpinski` qui rend l'image d'un triangle de Sierpinski.

Si vous essayez de dessiner un triangle de Sierpinski de rang élevé, le temps d'exécution risque d'être très long. Pour comprendre la raison de cette lenteur, déterminez le nombre d'abscisses et ordonnées à calculer pour dessiner un triangle de Sierpinski de rang n.

Question 2

On veut maintenant dessiner des triangles de Sierpinski ABC isocèles, de sommet A et dont la base $[BC]$ est horizontale. Écrire la *spécification* et une *définition* de la fonction `sierpinski-isocele` qui répond à ce problème.

Question 3

On veut maintenant dessiner des triangles de Sierpinski ABC équilatéraux, dont le côté $[BC]$ est horizontal. Écrire la *spécification* et une *définition* de la fonction `sierpinski-equilateral` qui répond à ce problème.

Chapitre 4

Liste et récursion simple

4.1 Constructions et manipulations simples de listes

Exercice 29. Expressions simples pour construire des listes

Le but de cet exercice est de donner des expressions simples qui permettent de construire une liste, de récupérer un élément particulier dans une liste et, enfin, de faire la différence entre les diverses primitives à votre disposition.

Question 1

Donner les expressions Scheme qui rendent les listes suivantes, ainsi que le type de ces listes :

1. `()`
2. `(1 2)`
3. `((1) (2))`
4. `((1 2) (3))`
5. `((1 2) (3 4) (5 6))`
6. `(())`

Question 2

Donner l'expression Scheme qui, appliquée à la liste `((1 2) (3 4) (5 6))`, rend :

1. la liste `(1 2)`
2. la liste `((3 4) (5 6))`
3. la liste `(3 4)`
4. la valeur `2`
5. la valeur `4`
6. la liste `((5 6))`
7. la liste `(5 6)`

Question 3

Donner le résultat de l'évaluation des expressions suivantes :

1. `(append (list 1 2) (list 3))`
2. `(cons (list 1 2) (list 3))`
3. `(list (list 1 2) (list 3))`

4. `(cons 3 (append (list 1) (list 2)))`
5. `(list (append (list 1 2) (list 3)) (cons 4 (list)))`
6. `(caddr (append (list (list 3 4)) (list 5 6 7)))`

Exercice 30. Fonction simple de manipulation de listes

Question 1

Donner la *signature* et une *définition* du semi-prédicat `rend-troisieme` qui, étant donnée une liste rend soit le troisième élément de cette liste s'il existe, soit faux, si la liste ne possède pas au moins trois éléments. Par exemple :
`(rend-troisieme (list 1 2 3))` → `3`
`(rend-troisieme (list 1 2))` → `#f`
`(rend-troisieme (list 43 22 42 77 987))` → `42`

Exercice 31. Liste des racines d'une équation du second degré

Le but de cet exercice est de calculer des racines d'une équation du second degré.

Question 1

En utilisant des *alternatives*, donner la *spécification* et une *définition Scheme* de la fonction `liste-racines` telle que `(liste-racines a b c)`, avec `a` non nul, retourne les racines de l'équation $ax^2 + bx + c = 0$ sous la forme d'une liste (contenant 0 ou 1 ou 2 éléments).
Par exemple :
`(liste-racines 1 2 3)` → `()`
`(liste-racines 1 2 1)` → `(-1)`
`(liste-racines 1 2 -3)` → `(1 -3)`

Question 2

Écrire une autre *définition* de cette fonction (que l'on nommera `liste-racines-cond`) en utilisant une *conditionnelle*.

Exercice 32. Fonction mystère

Le but de cet exercice est de trouver la spécification d'une fonction à partir de sa définition ainsi que d'évaluer pas à pas des fonctions.

Question 1

Voici la définition de la fonction `mystere1` :

```
(define (mystere1 x y)
  (if (pair? x)
      (if (= 0 (remainder (car x) y))
          (cons (car x) (mystere1 (cdr x) y))
          (mystere1 (cdr x) y))
      (list)))
```

En étudiant cette définition, déterminer le type des paramètres de cette fonction ainsi que le type de son résultat. Donner ensuite la spécification de cette fonction renommée avec un nom plus significatif, tout comme ses paramètres.

Question 2

Voici la définition de la fonction `mystere2` :

```
(define (mystere2 x y)
  (if (pair? x)
      (if (= 0 (remainder (car x) y))
          (+ (car x) (mystere2 (cdr x) y))
          (mystere2 (cdr x) y))
      0))
```

Donner le résultat de l'évaluation de l'application `(mystere2 (list 4 5 6) 2)` en donnant quelques étapes intermédiaires. En déduire la spécification de la fonction.

4.2 Construction de listes avec récursion sur entier

Exercice 33. Liste de répétitions

Petite fonction simple de construction de liste.

Question 1

Donner la *spécification* et une *définition* de la fonction `repete` qui étant donnés un naturel n et un nombre x construit la liste contenant n occurrences de x. Par exemple :
`(repete 0 3)` → `()`
`(repete 3 3)` → `(3 3 3)`
`(repete 5 0)` → `(0 0 0 0 0)`

Exercice 34. Intervalle d'entiers

Question 1

Donner la *spécification* et une *définition* de la fonction `intervalle` qui étant donnés deux nombres entiers a et b construit la liste contenant tous les entiers de a inclus à b inclus. On suppose que a est inférieur ou égal à b. Par exemple :
`(intervalle -2 3)` → `(-2 -1 0 1 2 3)`
`(intervalle 3 3)` → `(3)`
`(intervalle 7 10)` → `(7 8 9 10)`

Question 2

En mathématiques, par convention, l'intervalle $[a, b]$ est vide lorsque a est strictement supérieur à b. Donner la *signature* et une *définition* de la fonction `intervalle-gen` telle que `(intervalle-gen a b)` renvoie la liste des entiers compris entre a inclus et b inclus, sans faire l'hypothèse que $a \leq b$. Par exemple :
`(intervalle-gen -2 3)` → `(-2 -1 0 1 2 3)`
`(intervalle-gen 3 3)` → `(3)`
`(intervalle-gen 4 3)` → `()`
`(intervalle-gen 2 -2)` → `()`

4.3 Récursion sur liste calculant une valeur

Exercice 35. Sommes sur les éléments d'une liste

Le but de cet exercice est de calculer un nombre à partir d'une liste de nombres : la somme des éléments, la somme des carrés des éléments et la moyenne des éléments.

Question 1

Donner la *signature* et une *définition* de la fonction `somme-liste` qui, étant donnée une liste de nombres, rend la somme des éléments de cette liste. Par convention, la somme des éléments d'une liste vide est 0. Par exemple :
`(somme-liste (list 2 5 7 3))` → `17`
`(somme-liste (list))` → `0`

Question 2

Donner la *signature* et une *définition* de la fonction `somme-liste-carres` qui, étant donnée une liste de nombres, rend la somme des carrés des éléments de cette liste. Par convention, on rend 0 pour une liste vide. Par exemple :
`(somme-liste-carres (list 2 5 7 3))` → `87`
`(somme-liste-carres (list))` → `0`

Question 3

Donner la *spécification* et une *définition* de la fonction `moyenne-liste` qui retourne la moyenne des éléments d'une liste de nombres. Cette fonction devra signaler une erreur lorsque la liste est vide.
Par exemple :
`(moyenne-liste (list 4 12 8))` → `8`
`(moyenne-liste (list))` → `erreur "moyenne-liste : liste vide"`

Exercice 36. Autour de la longueur d'une liste

Question 1

Donner la *signature* et une *définition* de la fonction `longueur` qui étant donnée une liste renvoie son nombre d'éléments. Par exemple :
`(longueur (list 1 2 3 4))` → `4`
`(longueur (list "a" "b" "c"))` → `3`
`(longueur (list))` → `0`

Question 2

Donner la *signature* et une *définition* du prédicat `longueur-paire?` qui étant donnée une liste renvoie vrai si et seulement si la liste comporte un nombre pair d'éléments. Par exemple :
`(longueur-paire? (list 1 2 3 4))` → `#t`
`(longueur-paire? (list "a" "b" "c"))` → `#f`
`(longueur-paire? (list))` → `#t`

Question 3

Pour vérifier qu'une liste comporte au moins trois éléments, on pourrait penser utiliser la fonction `longueur` et écrire la définition suivante :

```
;;; plus-de-3? : LISTE[alpha] -> bool
;;; (plus-de-3? L) rend vrai ssi la liste L comporte au moins trois éléments
(define (plus-de-3? L)
  (>= (longueur L) 3))
```

Cette solution n'est pas efficace, car la fonction `longueur` parcourt inutilement toute la liste (qui peut être très longue), alors que l'on veut seulement déterminer s'il y a plus de trois éléments.

Donner la *spécification* et une *définition* du prédicat `longueur-sup3?` qui, étant donnée une liste L, vérifie que la liste a au moins trois éléments, et ce, sans parcourir toute la liste.

Question 4

Donner la *signature* et une *définition* du prédicat `longueur-egale3?` qui, étant donnée une liste L, vérifie que la liste a exactement trois éléments.

Exercice 37. Présence d'un élément dans une liste

Le but de cet exercice est de tester la présence d'un élément dans une liste.

Question 1

Donner la *spécification* et une *définition* du prédicat `est-dans?` qui, étant donnés un élément e et une liste L rend vrai si et seulement si e est un élément de L. On utilisera ici des alternatives.
Par exemple :
`(est-dans? 3 (list 1 8 2 3 4 ))` → `#t`
`(est-dans? 8 (list 2 4 3 1 5))` → `#f`

Question 2

En utilisant uniquement des formes spéciales `or` et `and`, écrire une *définition* de la fonction `est-dans-bis?`, qui répond à la même spécification que la fonction `est-dans?`.

Exercice 38. Nombre d'occurrences d'un élément dans une liste

Le but de cet exercice est de calculer le nombre d'occurrences d'un élément dans une liste.

Question 1

Écrire la *spécification* et une *définition Scheme* de la fonction `nombre-occurrences` qui, étant donnés un élément e et une liste d'éléments L, renvoie le nombre d'occurrences de e dans L (par convention on rend 0 lorsque la liste est vide).
Par exemple :
`(nombre-occurrences 3 (list 1 3 2 3 6 5))` → `2`
`(nombre-occurrences 3 (list))` → `0`
`(nombre-occurrences "me" (list "me" "ma" "me" "meuh" "me"))` → `3`

Exercice 39. Tout ce qui n'est pas faux est vrai

Le but de cet exercice est d'écrire un semi-prédicat et de l'utiliser ensuite afin de voir l'intérêt d'un semi-prédicat sur un prédicat.

Rappel : un *semi-prédicat* est un presque un prédicat : c'est une fonction qui renvoie soit `#f`, soit la valeur vrai au travers d'une valeur quelconque différente de `#f`. Pour utiliser les semi-prédicat, il faut savoir qu'en *Scheme* tout ce qui n'est pas faux est vrai mais pas égal à `#t` sauf `#t` lui-même. Par exemple :
`(if 1 "vrai" "faux")` → `"vrai"`
`(if "faux" "vrai" "faux")` → `"vrai"`

`(equal? #t 1)` → `#f`
`(equal? #t "faux")` → `#f`
`(equal? #t #t)` → `#t`

Question 1

Donner la *signature* et une *définition* du semi-prédicat `sous-liste-ou-faux` qui étant donnés un élément e et une liste L, renvoie la plus grande sous-liste suffixe de L dont le premier élément est e, si e apparaît dans L et la valeur `#f` sinon. Par exemple :
`(sous-liste-ou-faux 1 (list 2 3 1 4 1))` → `(1 4 1)`
`(sous-liste-ou-faux 5 (list 2 3 1 4 1))` → `#f`
`(sous-liste-ou-faux "jo" (list "ema" "zoe" "jo"))` → `("jo")`

Question 2

Donner la *signature* et une *définition* d'un semi-prédicat qui, étant donnés un élément e et une liste L, renvoie l'élément suivant la première occurrence de e dans L, si e apparaît dans L et si la première occurrence de e dans L n'est pas en dernière position dans L, et renvoie `#f` sinon. Par exemple :
`(suivant 1 (list 1 4 6 3 2 1 7))` → `4`
`(suivant 2 (list 1 4 6 3 2 1 7))` → `1`
`(suivant 7 (list 1 4 6 3 2 1 7))` → `#f`
`(suivant 5 (list 1 4 6 3 2 1 7))` → `#f`

Exercice 40. Maximum d'une liste de nombres

Le but de cet exercice est de calculer le maximum d'une liste de nombres ainsi que le nombre d'occurrences du maximum d'une liste.

Question 1

Donner la *spécification* et une *définition Scheme* de la fonction `max-liste` qui, étant donnée une liste non vide de nombres, renvoie la valeur du maximum de la liste. Par exemple :
`(max-liste (list 1 3 8 5 7 8 2))` → `8`
`(max-liste (list -23))` → `-23`

Question 2

Donner la *spécification* et une *définition Scheme* de la fonction `nombre-de-max` qui, étant donnée une liste de nombres, renvoie le nombre d'occurrences du maximum de la liste. Par convention on renvoie 0 lorsque la liste est vide.
Par exemple :
`(nombre-de-max (list 1 8 2 8 6 5 8))` → `3`
`(nombre-de-max (list))` → `0`

Dans cette question on utilisera la fonction `nombre-occurrences` définies dans l'exercice **??** page **??** dont voici la spécification :

```
;;; nombre-occurrences : alpha * LISTE[alpha] -> nat
;;; (nombre-occurrences e L) rend le nombre d'occurrences de  e dans L.
;;; Par convention rend 0 si L est vide
```

Exercice 41. Variations autour du signe des éléments d'une liste

Le but de cet exercice est de compter le nombre d'éléments d'un certain signe ou le nombre de changements de signes dans une liste de nombres. Cet exercice est issu du devoir sur table de novembre 2006.

Question 1

Donner la *spécification* et une *définition* du prédicat `meme-signe?` qui, étant donnés deux entiers non nuls, teste si ces deux entiers sont de même signe. Par exemple :
`(meme-signe? -1 1)` → `#f`
`(meme-signe? 1 1)` → `#t`
`(meme-signe? -1 -1)` → `#t`

Question 2

Donner la *spécification* puis une *définition* de la fonction `nb-meme-signe` qui étant donnés L une liste d'entiers non nuls et n un entier non nul rend le nombre d'éléments de L qui ont le même signe que n. Par convention, la fonction rend 0 lorsque la liste est vide. Par exemple :
`(nb-meme-signe 3 (list -1 2 3 -5))` → `2`
`(nb-meme-signe -1 (list 2 3 -5 -6 -7))` → `3`

Question 3

Donner la *signature*, un *jeu de tests* de taille minimal et couvrant tous les cas puis une *définition* de la fonction `nb-chgt-signe` qui, étant donnée une liste L d'entiers non nuls, rend le nombre de fois que deux entiers consécutifs de L n'ont pas le même signe. Par exemple :
`(nb-chgt-signe (list -1 2 -3 -4))` → `2`

Exercice 42. Liste croissante

Le but de cet exercice est de tester si une liste de nombres est croissante.
On dit qu'une liste (e_1 e_2 ... e_n) est croissante (au sens large) lorsque $e_1 \leq e_2 \leq ... \leq e_n$.

Question 1

Écrire la *signature* et une *définition* récursive du prédicat `croissante?` qui teste si une liste de nombres est croissante au sens large. Notons que la liste vide et les listes réduites à un seul nombre sont croissantes. Par exemple :
`(croissante?(list 1 2 4 4 6 8))` → `#t`
`(croissante?(list 1 2 5 4 8))` → `#f`
`(croissante?(list 6))` → `#t`
`(croissante?(list))` → `#t`

Question 2

Écrire une autre *définition* récursive du prédicat `croissante?` en n'utilisant que des formes `or` et `and`. Cette fonction aura pour nom `croissante-bis?`

Exercice 43. Schéma de Hörner

Étant donné un polynôme $a_0+a_1.x+a_2.x^2+...+a_n.x^n$, représenté par la liste de ses coefficients ($a0$ $a1$ ... an), on veut évaluer ce polynôme pour une certaine valeur de x. Pour que l'évaluation soit efficace, on utilisera le schéma suivant, dit schéma de Hörner :

$$a_0 + a_1.x + a_2.x^2 + a_3.x^3 + a_4.x^4 = a_0 + x * (a_1 + x * (a_2 + x * (a_3 + x * a_4)))$$

Avec le schéma de Hörner, l'évaluation d'un polynôme de degré n nécessite n multiplications (et n additions). Lorsqu'une fonction calcule un résultat en un nombre d'opérations proportionnel à la *taille* des données, on dit que cette fonction est de *complexité linéaire en temps*. Ici la taille des données est $n+1$: les n coefficients et la valeur x. Le schéma de calcul de Hörner permet donc d'évaluer un polynôme en temps linéaire.

On a étudié des cas simples du schéma de Hörner dans l'exercice **??** page **??**.

Question 1

Écrire la *signature* et une *définition* de la fonction `horner`, qui reçoit un nombre et une liste de coefficients ($a0$ $a1$ ... an), et renvoie la valeur du polynôme en ce nombre. Par exemple :
`(horner 3 (list 1 3 0 5 0 1))` → `388`

4.4 Récursion sur liste construisant une liste

Exercice 44. Liste des carrés d'une liste

Le but de cet exercice est de construire la liste des carrés des éléments d'une liste de nombres.

Question 1

Écrire la *signature* et une *définition* de fonction *récursive* de nom `liste-carres` qui renvoie la liste des carrés des éléments d'une liste de nombres. Par exemple :
`(liste-carres(list 1 8 2 4 6 5 3))` → `(1 64 4 16 36 25 9)`

Exercice 45. Liste des éléments plus grands qu'un nombre donné

Le but de cet exercice est de construire la liste des éléments plus grands qu'un nombre donné dans une liste de nombres. Cet exercice est tout à fait analogue à l'exercice **??** page **??**.

Question 1

Donner la *spécification* et une *définition* récursive de la fonction `plus-grands` qui, étant donnés un nombre e et une liste de nombres L, renvoie la liste des éléments de L supérieurs ou égaux à e. Par exemple :
`(plus-grands 3 (list 1 8 2 4 5 5 3))` → `(8 4 5 5 3)`
`(plus-grands 8 (list 2 4 3 1 5))` → `()`

Exercice 46. Enlever toutes les occurrences d'un élément

Le but de cet exercice est de supprimer toutes les occurrences d'un élément donné dans une liste. Cet exercice est tout à fait analogue à l'exercice **??** page **??**

Question 1

Écrire la *signature* et une *définition* de la fonction `moins-occurrences` qui, étant donnés un élément *elt* et une liste L, renvoie la liste privée de toutes les occurrences de cet élément. Par exemple :
`(moins-occurrences 3 (list 1 3 4 3 5 5 3))` → `(1 4 5 5)`
`(moins-occurrences 3 (list 2 4 1 5))` → `(2 4 1 5)`
`(moins-occurrences "ma" (list "ma" "me" "ma" "mi" "mo" "ma" ))` → `("me" "mi" "mo")`

Exercice 47. Begaie — debegaie

On dit qu'une liste est « bégayée » lorsque chaque élément y apparaît deux fois de suite. Le but de cet exercice est de transformer une liste en une liste bégayée, et inversement de « débégayer » une liste en supprimant un élément sur deux.

Question 1

Écrire la *signature* et une *définition* de la fonction `begaie` qui, étant donnée une liste d'éléments, renvoie la liste où chaque élément est répété. Par exemple :
`(begaie (list 1 2 3 1 4))` → `(1 1 2 2 3 3 1 1 4 4)`
`(begaie (list))` → `()`

Question 2

Écrire la *signature* et une *définition* de la fonction `debegaie` qui, étant donnée une liste bégayée, restitue la liste initiale. C'est-à-dire que `(debegaie (begaie L))` = `L`. Par exemple :
`(debegaie (list 1 1 2 2 3 3 1 1 4 4))` → `(1 2 3 1 4)`
`(debegaie (begaie (list 1 2 3 1 4)))` → `(1 2 3 1 4)`

Question 3

Dans la question précédente on a supposé que la liste donnée était bégayée. On demande à présent d'écrire la *spécification* et une *définition* de la fonction `debegaie-verif` qui, étant donnée une liste L quelconque, signale une erreur lorsque L n'est pas bégayée, et sinon renvoie le « débégaiement » de L. Par exemple
`(debegaie-verif (list 1 1 2 2 3 3 1 1 4 4))` → `(1 2 3 1 4)`
`(debegaie-verif (list 1 1 2 ))` → `erreur "debegaie-verif : la liste donnée n'est pas une liste`

4.5 Double récursion et schéma de récursion avancée

Exercice 48. Listes des éléments d'une suite

Le but de cet exercice est de construire des listes contenant les éléments d'une suite définie récursivement. Nous construirons les listes dans les deux sens possibles : si u_n désigne un terme de la suite, alors nous pouvons construire la liste (u_n u_{n-1} ... u_0) ou la liste (u_0 u_1 ... u_n)

Question 1

Une suite géométrique $u_n, n \in N$ est définie récursivement par son premier terme u_0 et sa raison r par :
- $u_n = u_0$ si $n = 0$
- $u_n = u_{n-1} * r$ sinon

Donner la *signature* et une *définition* de la fonction `suivant` qui, étant donnés deux nombres u_n et r, rend le terme suivant de u_n dans la suite géométrique de raison r. Par exemple :
`(suivant 1 4)` → `4`
`(suivant 5 2)` → `10`

Question 2

Donner la *signature* et une *définition* de la fonction `liste0aN` qui, étant donnés deux nombres u_0 et r ainsi qu'un entier naturel n, renvoie la liste (u_0 u_1 ... u_n) où les u_i sont les éléments de la suite géométrique de raison r et de premier terme u_0. Par exemple :
`(liste0aN 1 2 5)` → `(1 2 4 8 16 32)`

`(liste0aN 1 2 0)` → `(1)`
`(liste0aN 3 1 4)` → `(3 3 3 3 3)`

Question 3

Donner la *signature* et une *définition* de la fonction `listeNa0` qui, étant donnés deux nombres u_0 et r ainsi qu'un entier naturel n, renvoie la liste $(u_n \ ... \ u_1 \ u_0)$ où les u_i sont les éléments de la suite géométrique de raison r et de premier terme u_0. Par exemple :
`(listeNa0 1 5 2)` → `(25 5 1)`
`(listeNa0 1 0 2)` → `(0 0 1)`
`(listeNa0 3 4 1)` → `(12 3)`

Pour répondre à cette question, vous n'utiliserez pas la fonction `liste0aN`

Question 4

La suite de Fibonacci est définie comme suit :
- $f_0 = 1$ et $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$ si $n > 1$

Donner la *signature* et une *définition* de la fonction `liste-fibo` qui étant donné un entier naturel n renvoie la liste $(f_n \ f_{n-1} \ f_{n-2} \ .... \ f_1 \ f_0)$ où les f_i sont les termes de la suite de Fibonacci. Par exemple :
`(liste-fibo 0)` → `(1)`
`(liste-fibo 1)` → `(1 1)`
`(liste-fibo 4)` → `(5 3 2 1 1)`
`(liste-fibo 5)` → `(8 5 3 2 1 1)`

Exercice 49. Recherche du n-ième élément d'une liste

Le but de cet exercice est de rechercher l'élément en n-ième position dans une liste.

Question 1

Donner la *spécification complète* et une *définition* de la fonction `n-ieme` qui, étant donnés un entier n et une liste, renvoie le n-ième élément de la liste. On numérotera les éléments d'une liste à partir de 0. On renverra un message d'erreur lorsque le traitement est impossible. Le message d'erreur indiquera pourquoi le traitement est impossible. Par exemple :
`(n-ieme 0 (list 2 3 5 7))` → `2`
`(n-ieme 4 (list 2 4 3 5 7 6))` → `7`
`(n-ieme 3 (list 8 2 6))` → `erreur "n-ieme : position trop grande"`
`(n-ieme -1 (list 8 2))` → `erreur "n-ieme : position doit être >=0"`

Indication : il est demandé de ne pas faire un premier parcours pour connaître la longueur de la liste puis un parcours pour rechercher le n-ième élément de la liste mais d'effectuer un seul parcours de la liste.

Exercice 50. Liste d'une certaine longueur

Dans cet exercice, on souhaite déterminer si une liste a une certaine longueur. La façon la plus simple est probablement d'utiliser la fonction `longueur` et d'écrire :

```
;;; lg-naif?: nat * LISTE[alpha] -> bool
;;; (lg-naif? n L) verifie que la liste L a pour longueur n.
(define (lg-naif? n l)
  (= n (longueur L)) )
```

Malheureusement calculer la longueur d'une liste a un coût proportionnel à la longueur de cette liste (on parle de coût linéaire). L'absurdité est alors de demander si une liste d'un million de termes a une longueur égale à zéro (ce qui peut se tester plus simplement et à coût constant avec le prédicat `pair?`).

Dans cet exercice, on veut donc déterminer si une liste a une certaine longueur n en appliquant au plus n fois la fonction `cdr`.

Question 1

Donner la *spécifiation* et une *définition* du prédicat `lg?` qui, étant donnés un entier naturel n et une liste L, vérifie si la liste est de taille n. Ainsi :
`(lg? 0 (list))` → `#t`
`(lg? 2 (list "a" "bb" "ccc"))` → `#f`
`(lg? 2 (list "a" "bb"))` → `#t`

Question 2

Donner des applications de la fonction `lg-fausse?` qui renvoient une erreur et expliquer pourquoi la définition de `lg-fausse?` n'est pas correcte.

```
(define (lg-fausse? n L)
  (if (> n 0)
      (lg-fausse? (- n 1) (cdr L))
      (not (pair? L)) ) )
```

Exercice 51. Dégonfler une liste

Cet exercice généralise l'exercice **??** page **??**. La troisième question fait appel à la notion d'accumulateur.

Question 1

Soit la fonction `a-deviner` suivante :

```
;;; a-deviner : ???
;;; (a-deviner y) rend ???
(define (a-deviner y)
  ;;; rept-ale : ???
  ;;; (rept-ale x) rend ???
  (define (rept-ale x)
    ;; rept-x : ???
    ;; (rept-x z) rend ???
    (define (rept-x z)
      (if (> z 0)
          (cons x (rept-x (- z 1)))
          (list)))

    (let ((t (+ 1 (random 9))))
      (rept-x t)))
  ; debut de a-deviner
  (if (pair? y)
      (append (rept-ale (car y)) (a-deviner (cdr y)))
      (list)))
```

La fonction `(random k)`, rend un nombre entier aléatoire compris entre 0 et $k-1$.

Donner le résultat de l'application `(a-deviner(list 1 2 1 4))` puis les spécifications des fonctions internes et de la fonction `a-deviner` en donnant des noms plus significatifs aux fonctions et à leurs paramètres.

Question 2

On veut maintenant « dégonfler » une liste. Écrire la *signature* et une *définition* de la fonction `degonfle` qui étant donnée une liste L, rend la liste dans laquelle toutes les occurences consécutives d'un même élément dans L sont remplacées par une seule. Ainsi lorsque L ne contient pas 2 occurences consécutives d'un même élément, on a `(degonfle L) = L`. Par exemple :
`(degonfle (list 1 1 1 1 2 2 2 1 1 4 4 4 4 4))` → `(1 2 1 4)`
`(degonfle (gonfle (list 1 2 1 4)))` → `(1 2 1 4)`
`(degonfle (gonfle (list 1 1 4)))` → `(1 4)`

Question 3

Écrire la *signature* et une *définition* de la fonction `degonfle-renverse`, qui, étant donnée une liste L, rend la liste en sens inverse, dans laquelle toutes les occurences multiples successives sont remplacées par une seule. Par exemple :
`(degonfle-renverse (list 1 1 1 1 2 2 2 1 1 4 4 4 44))` → `(44 4 1 2 1)`
`(degonfle-renverse (gonfle (list 1 2 4 4 1 4)))` → `(4 1 4 2 1)`

Indication : on utilisera une fonction interne auxiliaire qui aura un paramètre supplémentaire permettant d'accumuler les résultats.

Exercice 52. Tri fusion

Le but de cet exercice est d'implanter une fonction de tri reposant sur un algorithme classique : le *tri par fusion.*

Question 1

Écrire la *signature* et une *définition* de la fonction `interclassement` qui, à partir de deux listes de nombres, croissantes au sens strict, construit la liste, croissante au sens strict, résultant de l'interclassement des deux listes initiales.
Attention : le résultat ne doit pas contenir de doublons. Par exemple :

`(interclassement '(2 4 6 8) '(1 3 4 5 7))` → `(1 2 3 4 5 6 7 8)`
`(interclassement '(1 3 5) '(2 4))` → `(1 2 3 4 5)`
`(interclassement '() '(2 4 5 7 9))` → `(2 4 5 7 9)`

Question 2

Écrire les *signatures* et les *définitions* des deux fonctions `pos-paires` et `pos-impaires` qui, à partir d'une liste d'éléments de type quelconque, construisent chacune une liste :

- `pos-paires` rend la liste des éléments en position paire dans la liste initiale,
- `pos-impaires`, la liste des éléments en position impaire dans la liste initiale.

Attention : par convention, le premier élément de la liste est en position 0 (considérée comme paire), le deuxième en position 1 ...

`(pos-paires '(a b c d e f g h))` →`(a c e g)`
`(pos-impaires '(a b c d e f g h))` →`(b d f h)`
`(pos-paires '(12))` →`(12)`
`(pos-impaires '(12))` →`()`

Question 3

En déduire la *spécification* et une *définition* de la fonction `tri-fusion` qui, étant donnée une liste de nombres retourne la liste de ces nombres en ordre strictement croissant (chaque nombre de la liste initiale n'apparaît qu'une fois).

Le principe du tri par fusion est le suivant : si la liste donnée est vide ou réduite à un élément, on la renvoie telle quelle ; sinon on la sépare en deux sous-listes -la sous-liste des éléments en position paire et la sous-liste des éléments en position impaire- le résultat recherché est alors l'interclassement du tri de chacune de ces deux sous-listes.

```
(tri-fusion '(5 1 8 9 1 2 4 3 10 13 1 6 11 7))
→(1 2 3 4 5 6 7 8 9 10 11 13)
(tri-fusion '(1 1 1 1)) → (1)
```

Question 4

Dans la définition de la fonction `tri-fusion`, on parcourt deux fois la liste : une fois pour calculer la liste des éléments en position paire et une fois pour calculer la liste des éléments en position impaire. Pour éviter ce double parcours, on considère une fonction qui calcule ces deux listes en même temps.

Écrire la *signature* et une *définition* de la fonction `pos-paires-impaires` qui, étant donnée une liste `L` d'éléments de type quelconque, renvoie le couple dont le premier élément est la liste des éléments en position paire dans la liste `L` et dont le deuxième élément est la liste des éléments en position impaire dans la liste `L`. Par exemple :

```
(pos-paires-impaires '(a b c d e f g h)) →((a c e g) (b d f h))
(pos-paires-impaires '(a b c d e f g h i)) →((a c e g i) (b d f h))
(pos-paires-impaires '(a)) →((a) ())
(pos-paires-impaires '()) →(() ())
```

Question 5

Écrire une *définition* de la fonction `tri-fusion-mieux` qui utilise la fonction `pos-paires-impaires` pour réaliser le tri par fusion d'une liste de nombres.

Chapitre 5

Couples, n-uplets et listes avancées

5.1 Couples et n-uplets

Exercice 53. N-uplets simples

Dans tout cet exercice, on souhaite manipuler des coordonnées de points dans le plan sous la forme d'un `COUPLE[Nombre Nombre]`. Ainsi, un point sera représenté par un couple (abscisse, ordonnée) qui le repérera dans le plan. Par exemple, le point à l'origine des axes est représenté par le couple (0 0). Pour simplifier les notations, dans ce qui suit, on pose `POINT=COUPLE[Nombre Nombre]`.

Question 1

Écrire la *spécification* et une *définition* de la fonction `abscisse` qui, étant donné un `POINT` rend la valeur correspondant à l'abscisse de ce point. Par exemple :
`(abscisse (list 0.5 4.2))` → `0.5`

Question 2

Écrire la *spécification* et une *définition* de la fonction `ordonnee` qui, étant donné un `POINT` rend la valeur correspondant à l'ordonnée de ce point. Par exemple :
`(ordonnee (list 0.5 4.2))` → `4.2`

Question 3

On rappelle que, dans le plan, l'équation d'une droite est donnée par $y = ax + b$ ce qui permet, connaissant les paramètres a et b de la droite, de vérifier qu'un point d'abscisse x et d'ordonnée y appartient à la droite (si l'égalité est vérifiée).

Écrire la *spécification* et une *définition* du prédicat `estDansDroite?` qui, étant donnés un `POINT` `p`, et deux nombres réels `a` et `b`, rend vrai si `p` appartient à la droite d'équation $y = ax + b$.

Par exemple :
`(estDansDroite? (list 1.5 6) 2 3)` → `#t`

Question 4

Écrire la *spécification* et une *définition* de la fonction `donneCoefficients` qui, étant donnés deux points `p1` et `p2` rend le n-uplet composé par les valeurs `a` et `b` telles que $y = ax + b$ soit

l'équation de la droite passant par `p1` et `p2`. On supposera que les points `p1` et `p2` n'ont pas la même abscisse.

Par exemple :
`(donneCoefficients (list 1 1) (list 5 9))` → `(2 -1)`

Exercice 54. Triplets et liste de n-uplets

Dans cet exercice, on va utiliser diverses représentations d'une même donnée.

Un *nolife* décide de faire un programme Scheme pour gérer sa liste d'amis dans son MMORPG [1] préféré. Dans ce jeu, chaque ami d'un nolife gère un personnage qui se caractérise par son nom, sa classe de personnage (guerrier, voleur, mage, etc.), et son niveau (un nombre entier). Dans un premier temps, notre nolife décide d'utiliser un n-uplet à 3 éléments pour représenter ces informations. Ainsi la fiche d'un ami sera donc un `NUPLET[String Int String]` avec le nom, le niveau, et la classe de son personnage. Par exemple, les meilleurs amis de notre nolife sont : `("Gimli" 32 "guerrier")`, `("Gandalf" 55 "mage")`, `("Bilbo" 31 "voleur")`, et `("Legolas" 34 "chasseur")`.

Dans tout le reste de cet exercice, on pose `FICHESIMPLE = NUPLET[String Int String]` pour alléger les notations.

Question 1

Écrire les *spécifications* et des *définitions* des trois fonctions `rendNom`, `rendNiveau`, et `rendClasse` qui, étant donnée une `FICHESIMPLE`, rendent respectivement le nom, le niveau, et la classe du personnage.

Par exemple :
`(rendNom (list "Gandalf" 55 "mage"))` → `"Gandalf"`
`(rendNiveau (list "Gandalf" 55 "mage"))` → `55`
`(rendClasse (list "Gandalf" 55 "mage"))` → `"mage"`

Question 2

Les informations sur un ami de notre nolife peuvent évoluer au fil du temps. En particulier, un personnage peut augmenter son niveau (cela se réalise en rajoutant 1 au niveau du personnage), il faut donc une fonction qui permettra de modifier la fiche du personnage correspondant.

Écrire la *spécification* et une *définition* de la fonction `augmenteNiveau` qui, étant donnée une `FICHESIMPLE` rend cette fiche en augmentant le niveau de 1.

Par exemple :
`(augmenteNiveau (list "Gandalf" 55 "mage"))` → `("Gandalf" 56 "mage")`
`(augmenteNiveau '("Bilbo" 31 "voleur"))` → `("Bilbo" 32 "voleur")`

En fait, il apparaît que la représentation précédente possède quelques inconvénients. En particulier, l'ordre des valeurs dans le n-uplet est très important : il ne faut surtout pas intervertir le niveau et la classe par exemple. Pour pallier cet inconvénient, une nouvelle idée est de représenter chaque personnage par une liste de n-uplets à 2 éléments, le premier élément de ces n-uplet sera une catégorie (nom, classe, niveau,...), et le second élément sera la valeur associée. Ainsi, les amis de notre nolife seront alors représentés par les listes de n-uplets suivantes :
`(("nom" "Gimli")("niveau" 32)("classe" "guerrier"))`,
`(("nom" "Gandalf")("classe" "mage")("niveau" 55))`,
`(("nom" "Bilbo")("niveau" 31)("classe" "voleur"))`,
et `(("niveau" 34)("classe" "chasseur")("nom" "Legolas"))`,

1. *Massively Multiplayer Online Role-Playing Game* (ou *jeu de rôle en ligne massivement multijoueurs* en français).

Chaque information sur un personnage sera donc représentée dans un `NUPLET[String Valeur]`. Une fiche de personnage sera donc une liste qui sera soit vide, soit contenant des n-uplets donnant des informations sur le personnage.
Dans ce qui suit, on pose `FICHE = Liste[NUPLET[String Valeur]]`.

Question 3

Écrire la *spécification* et une *définition* de la fonction `rendNom-bis` qui, étant donnée une `FICHE`, rend le nom du personnage correspondant, ou faux si la fiche ne contient pas de nom. On prendra soin de faire attention au fait que le nom d'un personnage ne sera pas forcément en tête dans la liste.

Par exemple :
`(rendNom-bis '(("nom" "Gandalf") ("niveau" 55) ("classe" "mage")))` → `"Gandalf"`
`(rendNom-bis '(("niveau" 34) ("classe" "chasseur") ("nom" "Legolas")))` → `"Legolas"`
`(rendNom-bis '(("niveau" 70) ("classe" "démoniste")))` → `#f`

Question 4

Pour éviter de devoir écrire une fonction par catégorie à afficher, on décide d'écrire une fonction qui, en plus d'une fiche, prendra en argument un nom de catégorie à retrouver.

Écrire la *spécification* et une*définition* de la fonction `rendValeur` qui, étant données une catégorie (`"nom"`, `"niveau"`, `"classe"`, ou tout autre chaîne de caractères) et une `FICHE`, rend la valeur du champ correspondant, ou faux si la fiche ne contient pas de n-uplet associé à la catégorie donnée.

Par exemple :
`(rendValeur "nom" '(("nom" "Gimli")("niveau" 32)("classe" "guerrier")))` → `"Gimli"`
`(rendValeur "classe" '(("niveau" 34) ("nom" "Legolas") ("classe" "chasseur")))`
→`"chasseur"`
`(rendValeur "age" '(("nom" "Gandalf") ("niveau" 55) ("classe" "mage")))` → `#f`

Question 5

Écrire la *spécification* et une *définition* de la fonction `augmenteNiveau-bis` qui, étant donnée une `FICHE` rend cette fiche en augmentant le niveau de 1. Par exemple :
`(augmenteNiveau-bis '(("nom" "Gandalf") ("niveau" 55) ("classe" "mage")))`
→`(("nom" "Gandalf") ("niveau" 56) ("classe" "mage"))`
`(augmenteNiveau-bis '(("nom" "Bilbo") ("classe" "voleur") ("niveau" 31)))`
→`(("nom" "Bilbo") ("classe" "voleur") ("niveau" 32))`

5.2 Récursion et couple

Exercice 55. Division euclidienne

On illustre dans cet exercice comment une fonction peut calculer "plusieurs" valeurs et comment utiliser celles-ci.

Question 1

La division euclidienne de m par n calcule un quotient q *et* un reste r tels que $m = n \times q + r$. Sans utiliser les fonctions `quotient` et `remainder`, écrire une *définition* de la fonction récursive `quotient-et-reste` de spécification :

```
;;; quotient-et-reste: nat * nat -> COUPLE[nat nat]
;;; (quotient-et-reste m n) calcule le couple (q r) tel que m = n*q + r
;;; HYPOTHESE : n est non nul
```

et dont voici quelques exemples d'application :
`(quotient-et-reste 0 5)` → `(0 0)`
`(quotient-et-reste 3 5)` → `(0 3)`
`(quotient-et-reste 10 5)` → `(2 0)`
`(quotient-et-reste 11 5)` → `(2 1)`

Indications.

1. Lorsque $m < n$ c'est le cas de base, le quotient est 0 et le reste est m ;
2. Sinon, c'est-à-dire lorsque $m \geq n$, c'est le cas récursif, on utilise la construction `let` pour stocker le résultat de la division euclidienne de $m - n$ et n. On construit le résultat pour m et n en ajoutant 1 au premier élément de la liste obtenue pour $m - n$ et n et en gardant tel quel le second.

Question 2

Donner la *spécification* et une *définition* du prédicat `verif-euclide?` qui vérifie que les résultats obtenus avec la fonction définie à la question précédente sont les mêmes que ceux que l'on obtient avec les fonctions `quotient` et `remainder` de Scheme.

Exercice 56. Retour sur la moyenne d'une liste de nombres

Question 1

Dans l'exercice ?? page ?? nous avons défini la fonction `moyenne-liste` comme suit :

```
;;; moyenne-liste : LISTE[Nombre] -> Nombre
;;; (moyenne-liste L) rend la moyenne des termes de L
;;; ERREUR lorsque L est vide
(define (moyenne-liste L)
  (if (pair? L)
      (/ (somme-liste L) (longueur L))
      (erreur "moyenne-liste : liste vide")))
```

Dans cette définition, le calcul de la moyenne est réalisé en parcourant deux fois la liste lorsque la liste est non vide : une fois pour faire la somme et une fois pour calculer la longueur. Cette définition n'est pas efficace car on peut tout à fait parcourir une seule fois la liste en calculant en même temps la somme des éléments et la longueur de la liste.
Écrire une *définition* de la fonction `somme-longueur-liste` qui étant donnée une liste de nombres calcule en même temps sa somme et sa longueur et rend ces deux valeurs dans un couple. Voici la spécification de cette fonction :

```
;;; somme-longueur-liste : LISTE[Nombre] -> COUPLE[Nombre Nombre]
;;; (somme-longueur-liste L) rend le couple formé par la somme et
;;; la longueur des termes de L (et rend la liste (0 0) lorsque L est vide).
```

Par exemple :
`(somme-longueur-liste (list 4 12 8))` → `(24 3)`
`(somme-longueur-liste (list))` → `(0 0)`

Question 2

Écrire une *définition* de la fonction `moyenne-liste-bis`, de même spécification que la fonction `moyenne-liste`, qui fait appel à la fonction `somme-longueur-liste`.

Exercice 57. Nombre d'occurrences du maximum d'une liste

Dans l'exercice **??** page **??**, nous avons défini la fonction `nombre-de-max` de spécification :

```
;;; nombre-de-max : LISTE[Nombre] -> nat
;;; (nombre-de-max L) rend le nombre d'occurrences du maximum de la liste L.
;;; Par convention rend 0 lorsque L est vide.
```

La définition que nous avons proposée n'est pas efficace parce qu'elle effectue deux parcours de la liste :

- un parcours effectué par la fonction `max-liste` (définie dans le même exercice) qui calcule le maximum d'une liste de nombres,
- et un autre parcours effectué par la fonction `nombre-d'occurrences` (voir exercice **??** page **??**) pour calculer le nombre d'occurrences du maximum trouvé.

Le but de cet exercice est de calculer le nombre d'occurrences du maximum dans une liste de nombres en un seul parcours de liste.

Question 1

Pour calculer le nombre d'occurrences du maximum d'une liste de manière efficace, c'est-à-dire en un seul parcours de la liste, on va définir une fonction `qui-combien`, qui étant donnée une liste non vide de nombres rend le couple formé du maximum de la liste et de son nombre d'occurrences :

```
;;; qui-combien :  LISTE[Nombre] -> NUPLET[Nombre  nat]
;;; (qui-combien L) rend le doublet formé du maximum de L et de son
;;; nombre d'occurrences
;;; HYPOTHESE : L est non vide
```

Par exemple :
`(qui-combien (list 1 8 2 8 6 5 8))` → `(8 3)`
`(qui-combien (list 3))` → `(3 1)`

Écrire une *définition* de la fonction `qui-combien`.

Question 2

Écrire une *définition* de la fonction `nombre-de-max-bis`, de même spécification que la fonction `nombre-de-max`, qui fait appel à `qui-combien` pour calculer le nombre d'occurrences du maximum dans une liste de nombres.

5.3 Liste de listes

Exercice 58. Liste de listes

Le but de cet exercice est d'écrire des fonctions qui manipulent ou construisent des listes de listes.

Question 1

Donner la *signature* et une *définition* de la fonction `liste-des-suffixes` qui étant donnée une liste L renvoie la liste des sous-listes suffixes de L. Par exemple :
`(liste-des-suffixes (list "luc" "eve" "jo"))` →`(("luc" "eve" "jo") ("eve" "jo") ("jo") ())`
`(liste-des-suffixes (list 1 2 3 4))` → `((1 2 3 4) (2 3 4) (3 4) (4) ())`
`(liste-des-suffixes (list))` → `(())`

Question 2

Donner la *signature* et une *définition* de la fonction `liste-sup` qui étant donnés un nombre x et une liste de listes de nombres L renvoie la liste contenant les éléments de L dont le deuxième élément est supérieur ou égal à x. Par exemple :
`(liste-sup 3 '((1 2 3) (1 3 2) (3 4) (3) ()))` → `((1 3 2) (3 4))`
`(liste-sup 5 (list))` → `()`

5.4 Liste de n-uplets

Exercice 59. Fréquences

Le but de cet exercice est d'étudier la fréquence (nombre d'occurrences) des éléments d'une liste. A partir d'une liste, on construit une liste d'associations dans laquelle chaque élément de la liste est associé à son nombre d'occurrences dans la liste.

Question 1

Écrire la *signature* et une *définition* de la fonction `augmentation`, qui, étant donnés un élément et une liste d'associations de fréquences, renvoie la liste d'associations initiale dans laquelle la fréquence de l'élément a été augmentée de 1. Lorsque l'élément donné `a` n'était pas présent dans la liste d'associations d'origine, on rajoute à cette dernière une association `(a 1)`. Par exemple

`(augmentation 'a '())` → `((a 1))`
`(augmentation 'a '((b 1) (c 1)))` → `((b 1) (c 1) (a 1))`
`(augmentation 'c '((b 1) (c 1) (a 1)))` → `((b 1) (c 2) (a 1))`

Question 2

Donner la *signature* et en utilisant la fonction `augmentation` écrire une *définition* de la fonction `frequence` qui, étant donnée une liste d'éléments, renvoie la liste d'associations correspondant aux fréquences d'apparition de chaque élément dans la liste.

`(frequence '(a b a b c b))` → `((b 3) (c 1) (a 2))`

Question 3

On a déjà calculé le nombre d'occurrences d'un élément dans une liste dans l'exercice **??** page **??**. On demande ici de le faire en utilisant la fonction `frequence`.

Donner une *définition*, utilisant la fonction `fréquence`, de la fonction `nbre-occ` qui étant donnés un élément et une liste d'éléments, renvoie le nombre d'occurences de l'élément dans la liste. Par exemple :
`(nbre-occ 'a '(a b a c d a b))` → `3`
`(nbre-occ 'a '(c d b b b))` → `0`
Conseil : Penser à utiliser la primitive `assoc`.

Question 4

Écrire la *signature* et une *définition* du prédicat `tous-plus-frequents?` qui, étant donnés un nombre et une liste d'associations de fréquences, teste si tous les éléments ont une fréquence supérieure ou égale au nombre donné. Par exemple :

`(tous-plus-frequents? 5 '())` →`#t`
`(tous-plus-frequents? 3 '((b 3) (c 4) (a 2)))` →`#f`
`(tous-plus-frequents? 2 (frequence '(a b d a c d a b c)))` →`#t`

Question 5

Écrire la *signature* et une *définition* de la fonction `liste-plus-frequents` qui, étant donnés un nombre et une liste d'associations de fréquences, renvoie la liste d'associations des éléments dont la fréquence est supérieure ou égale au nombre donné.

```
(liste-plus-frequents 3 '((b 3) (c 4) (a 2)))
→((b 3) (c 4))

(liste-plus-frequents 2 (frequence '(a b d a a b)))
→((b 2) (a 3))
```

Exercice 60. Liste de n-uplets

Le but de cet exercice est d'écrire des fonctions qui manipulent des listes de n-uplets.

Question 1

Donner une *définition* de la fonction `N-max` dont la spécification est :

```
;;; N-max : LISTE[NUPLET[String Nombre Nombre]]
;;;                          -> NUPLET[String Nombre Nombre]
;;; (N-max L) renvoie le premier n-uplet de la liste L qui a le plus grand
;;; deuxième élément
;;; HYPOTHESE : L est non vide
```

Par exemple :

```
(N-max '(("bob" 2 10) ("leo" 12 13) ("bea" 15 13) ("luc" 4 9) ("eve" 15 16) ("jo"  14 11)))
→("bea" 15 13)
```

Question 2

Donner une *définition* de la fonction `N-superieurs` dont la spécification est :

```
;;; N-superieurs : Nombre*LISTE[NUPLET[String Nombre Nombre]]
;;;                                        -> LISTE[NUPLET[String Nombre Nombre]]
;;; (N-superieurs x L) renvoie la liste des n-uplets de la liste L dont le
;;;  deuxième élément est supérieur ou égal à x
```

Par exemple :

```
(N-superieurs 10 '(("bob" 2 10) ("leo" 12 13) ("bea" 15 13) ("luc" 4 9) ("eve" 15 16) ("jo"  14 1
→(("leo" 12 13) ("bea" 15 13) ("eve" 15 16) ("jo" 14 11))

(N-superieurs 13 '(("bob" 2 10) ("leo" 12 13) ("bea" 15 13) ("luc" 4 9) ("eve" 15 16) ("jo"  14 1
→(("bea" 15 13) ("eve" 15 16) ("jo" 14 11))
```

Question 3

Que faudrait-il changer dans votre définition pour renvoyer le dernier n-uplet de la liste dont le deuxième élément est le plus grand au lieu du premier ?

Question 4

Donner la *spécification* et une *définition* de la fonction `N-liste-premier` qui étant donnée une liste de n-uplets L renvoie la liste contenant le premier élément de chaque n-uplet de L. Par exemple :

```
(N-liste-premier  '(("bob" 2 10) ("leo" 12 13) ("bea" 15 13) ("luc" 4 9) ("eve" 15 16)))
→("bob" "leo" "bea" "luc" "eve")
```

```
(N-liste-premier  '((4 2 10) (4 12 13) (8 15 13) (10 4 9) (17 15 16)))
→(4 4 8 10 17)
```

Chapitre 6

Fonctionnelles

Dans ce chapitre on demande d'écrire des définitions *non récursives*, qui utiliseront les fonctionnelles `filter`, `map`, `reduce`.

Exercice 61. Exercices autour de filter

Le but de cet exercice est d'utiliser la fonctionnelle `filter` pour définir des fonctions déjà vues dans les exercices précédents. On rappelle la spécification de `filter` :

```
;;; filter: (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
;;; (filter p? L) filtre la liste L en ne conservant que les éléments
;;; pour lesquels p? retourne #t
```

Dans toutes les questions de cet exercice, on devra définir des prédicats internes aux fonctions que l'on veut redéfinir, de façon à pouvoir utiliser la fonctionnelle `filter`.

Question 1

Dans l'exercice **??** page **??**, nous avons défini la fonction `moins-occurrences` qui supprime toutes les occurrences d'un élément donné dans une liste.

Donner la *signature* et une nouvelle *définition* de la fonction `moins-occurrences` qui, étant donnés un élément *elt* et une liste L, renvoie la liste privée de toutes les occurrences de cet élément. On nommera cette nouvelle définition `moins-occurrences-bis` et on utilisera la fonctionnelle `filter`. Par exemple :
`(moins-occurrences-bis 3 (list 1 3 4 3 5 5 3))` → `(1 4 5 5)`
`(moins-occurrences-bis 3 (list 2 4 1 5))` → `(2 4 1 5)`
`(moins-occurrences-bis "ma" (list "ma" "me" "ma" "mi" "mo" "ma" ))` →`("me" "mi" "mo")`

Indication : pour pouvoir utiliser `filter` ici, il faut définir un prédicat, interne à la fonction `moins-occurrences-bis`, qui étant donné un élément x, renvoie vrai ssi x est différent de *elt*.

Question 2

Dans l'exercice **??** page **??**, nous avons défini la fonction `plus-grands` dont la spécification est :

```
;;;  plus-grands : Nombre * LISTE[Nombre] -> LISTE[Nombre]
;;;  (plus-grands e L) rend la liste des éléments de L supérieurs ou égaux à e
```

.

Donner la *signature* et une *définition* de la fonction `plus-grands-bis` de même spécification que la fonction `plus-grands` en utilisant la fonction `filter`. On rappelle que, étant donnés un

nombre e et une liste de nombres L, la fonction `plus-grands` renvoie la liste des éléments de L supérieurs ou égaux à e.

Question 3

Écrire la *signature* et une *définition* de la fonction `liste-plus-frequents` qui, étant donnés un nombre et une liste d'associations de fréquences (couple dont la deuxième composante est un entier, voir exercice ?? page ??), renvoie la liste d'associations des éléments dont la fréquence est supérieure ou égale au nombre donné. Votre définition doit utiliser la fonctionnelle `filter`, la version sans `filter` est l'objet de la dernière question de l'exercice ?? page ??. Par exemple :
`(liste-plus-frequents 3 '((b 3) (c 4) (a 2)))` →`((b 3) (c 4))`
`(liste-plus-frequents 5 '((b 3) (c 4) (a 2)))` →`()`
`(liste-plus-frequents 4 '((e 1) (d 8) (b 3) (c 4) (a 2)))` →`((d 8) (c 4))`

Question 4

Donner la *signature* et une *définition* de la fonction `liste-sup` qui étant donnés un nombre x et une liste de listes de nombres L renvoie la liste contenant les éléments de L dont le deuxième élément est supérieur ou égal à x. Par exemple :
Votre définition doit utiliser la fonctionnelle `filter`, la question sans fonctionnelle est posée dans l'exercice ?? page ?? :

Question 5

Donner une *définition* de la fonction `N-sup` dont la spécification est :

```
;;; N-sup : Nombre * LISTE[NUPLET[String Nombre Nombre]]
;;;                                    -> LISTE[NUPLET[String Nombre Nombre]]
;;; (N-sup x L) renvoie la liste des n-uplets de la liste L dont le deuxième
;;;  élément est supérieur ou égal à x
```

. Votre définition doit utiliser la fonctionnelle `filter`, une version sans fonctionnelle est demandée dans l'exercice ?? page ??.

Exercice 62. Exercices autour de la fonctionnelle map

Le but de cet exercice est d'utiliser la fonctionnelle `map` dont voici la spécification :

```
;;; map: (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
;;; (map f L) applique la fonction f aux éléments de la liste L
```

Dans toutes les questions de cet exercice, on devra définir des prédicats internes aux fonctions que l'on veut redéfinir, de façon à pouvoir utiliser la fonctionnelle `map`.

Question 1

Écrire une *définition* de la fonction `liste-carres-bis`, de même spécification que `liste-carres`, en utilisant la fonctionnelle `map`.

Question 2

Donner la *signature* et une *définition* de la fonction `somme-carres` qui étant donnée une liste de nombres renvoie la somme des carrés des éléments de la liste. Dans cette définition, la fonction `somme-liste` définie dans l'exercice ?? page ?? et la fonctionnelle `map` seront utilisées. Par exemple :
`(somme-carres(list 2 5 7 3))` → `87`

`(somme-carres(list))` → `0`

Question 3

Donner la *signature* et une *définition*, utilisant la fonctionnelle `map`, de la fonction `xmul-liste` qui étant donnée une liste de nombres L et un nombre x, renvoie la liste des éléments de L multipliés par x. Par exemple :
`(xmul-liste (list 1 2 3 4 5) 3.0)` → `(3.0 6.0 9.0 12.0 15.0)`
`(xmul-liste (list 100 50 25 12 1) 2)` → `(200 100 50 24 2)`

Question 4

Étant donnée une liste d'associations L écrire deux fonctions :
– la fonction `liste-clefs` qui rend la liste des clefs ;
– la fonction `liste-valeurs` qui rend la liste des valeurs ;
Exemples :

`(liste-clefs '((1 un) (2 deux) (3 trois)))` → `(1 2 3)`
`(liste-valeurs '((1 un) (2 deux) (3 trois)))` → `(un deux trois)`

Question 5

Étant données une liste de clefs Lc et une liste de valeurs Lv, de même longueur, écrire une fonction `liste-clefs-valeurs` qui rend la liste d'associations constituées par les clefs de Lc et les valeurs de Lv.
Exemples :

`(liste-clefs-valeurs '(1 2 3) '(un deux trois))` → `((1 un) (2 deux) (3 trois))`

Question 6

Donner la *signature* et une *définition*, utilisant la fonctionnelle `map`, de la fonction `mention-aliste` qui, étant donnée une liste d'associations L associant à un nom d'étudiant sa note, retourne la liste d'associations associant les noms d'étudiant de L avec leur mention. Les mentions seront calculées comme dans l'exercice **??** page **??**. Par exemple :
`(mention-aliste '(("Durand" 8.5) ("Dupond" 11) ("Morin" 12.5)))`
→`(("Durand" "Eliminé") ("Dupond" "Passable") ("Morin" "Assez Bien"))`
`(mention-aliste '(("Benichou" 14.5) ("Moreau" 18.5)))`
→`(("Benichou" "Bien") ("Moreau" "Très Bien"))`

Question 7

Donner la *spécification* et une *définition* de la fonction `N-liste-premier` qui étant donnée une liste de n-uplets L renvoie la liste contenant le premier élément de chaque n-uplet de L. Par exemple :
`(N-liste-premier '(("bob" 2 10) ("leo" 12 13) ("bea" 15 13) ("luc" 4 9) ("eve" 15 16)))`
→`("bob" "leo" "bea" "luc" "eve")`
`(N-liste-premier '((4 2 10) (4 12 13) (8 15 13) (10 4 9) (17 15 16)))`
→`(4 4 8 10 17)`
Il est demandé ici d'utiliser la fonctionnelle `map`, une définition sans fonctionnelle est demandée dans l'exercice **??** page **??**.

Exercice 63. Exercices avec reduce

Le but de cet exercice est d'utiliser la fonctionnelle `reduce` pour traiter différents problèmes sur les listes : calculer le maximum, la somme, le nombre d'occurrences du maximum, *etc* d'une liste de nombres.
Rappel : la fonction `reduce` a pour spécification :

```
;;; reduce: (alpha * beta -> beta) * beta * LISTE[alpha] -> beta
;;; (reduce f en L) réduit la liste L en appliquant la fonction
;;; binaire f. La valeur en est l'élément neutre pour f.
```

Dans toutes les questions de cet exercice, on devra définir des prédicats internes aux fonctions que l'on veut redéfinir, de façon à pouvoir utiliser la fonctionnelle `reduce`.

Question 1

En utilisant la fonction `reduce`, donner une *définition* des fonctions `somme` et `somme-carres`, qui rendent respectivement la somme et la somme des carrés des éléments d'une liste de nombres.

Ces questions ont été traitées sans utiliser `reduce` dans l'exercice **??** page **??**.

Question 2

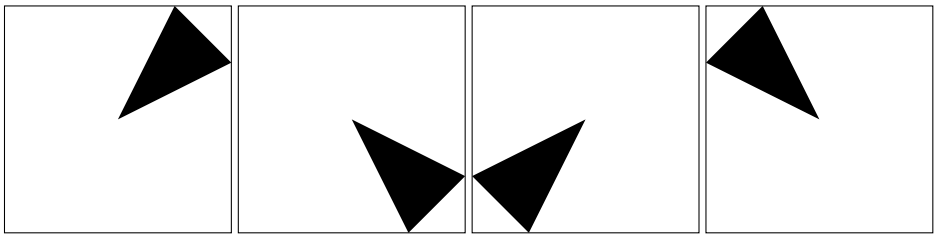
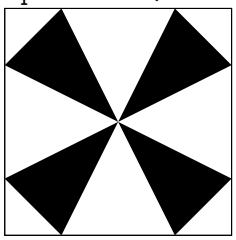
En utilisant la fonction `reduce`, donner une *définition* de la fonction `max-liste`, qui rend le maximum d'une liste non vide de nombres. Cette fonction a aussi été définie dans l'exercice **??** page **??** sans utiliser la fonction `reduce`

Question 3

En utilisant la fonction `reduce`, donner une nouvelle *définition* de la fonction `nombre-occurrences`, qui rend le nombre d'occurences d'un élément donné dans une liste. Voir l'exercice **??** page **??** pour des exemples d'utilisation.

Question 4

Donner la *signature* et une *définition*, utilisant `reduce`, de la fonction `superposition` qui étant donnée une liste d'images L rend l'image résultant de la superposition de toutes les images de L. Par exemple :

`(superposition (list (`  `) →` 

Exercice 64. Retour sur la croissance d'une liste

Le but de cet exercice est de tester la croissance d'une liste, en utilisant d'autres techniques que celle vue dans l'exercice **??** page **??**.

Question 1

Donner une *définition* de la fonction `inf` qui a la spécification suivante

```
;;; inf : Nombre * (Nombre+#f) -> (Nombre+#f)
;;; (inf x y) rend x si y est un nombre et x est inférieur ou égal à y,
;;; rend #f sinon
```

Par exemple
`(inf 2 1)` → `#f`
`(inf 2 2)` → `2`
`(inf 2 3)` → `2`
`(inf 2 #f)` → `#f`

Question 2

En utilisant la fonction `reduce` et la fonction `inf` de la question précédente, donner la *signature* et une *définition* de la fonction `croissante-reduce?` qui teste si une liste de nombres est croissante au sens large. On fait l'hypothèse que tous les nombres de la liste sont inférieurs à 1000.

Question 3

Donner une *définition* de la fonction `combine` qui a la spécification suivante

```
;;; combine : (alpha -> alpha) * LISTE[alpha] -> alpha
;;; (combine f L) rend (f e1 (f e2 ...(f eN-1 eN) ...))
;;; si L est la liste non vide (e1 e2 ... eN).
;;; ERREUR lorsque la liste est  vide
```

Question 4

En utilisant la fonction `combine`, donner une définition de la fonction `croissante-combine?` qui teste si une liste de nombres est croissante au sens large.

Question 5

Donner une *définition* de la fonction `inf2` qui a la spécification suivante :

```
;;; inf2 : (Nombre+#f) * Nombre -> Nombre+#f
;;; (inf2 x y) rend y si x est un nombre, inférieur ou égal à y,
;;; et sinon rend #f
```

Voici des exemples :
`(inf2 4 4)` → `4`
`(inf2 4 5)` → `5`
`(inf2 4 3)` → `#f`
`(inf2 #f 5)` → `#f`

Question 6

Écrire une fonction `croissante-accumulee?`, qui teste si une liste de nombres est croissante au sens large. On utilisera une fonction interne auxiliaire avec un paramètre permettant d'accumuler les résultats.

Exercice 65. Schéma de Hörner

Étant donné un polynôme $a_0+a_1.x+a_2.x^2+...+a_n.x^n$, représenté par la liste de ses coefficients `(a0 a1 ... an)`, on peut évaluer efficacement ce polynôme pour une certaine valeur de x en utilisant le schéma de Hörner reposant sur l'égalité suivante :

$$a_0 + a_1.x + a_2.x^2 + a_3.x^3 + a_4.x^4 = a_0 + x * (a_1 + x * (a_2 + x * (a_3 + x * a_4)))$$

On a étudié des cas simples du schéma de Hörner dans l'exercice **??** page **??** et proposé une définition dans l'exercice **??** page **??** de la fonction `horner` dont la spécification est :

```
;;;  horner : Nombre * LISTE[Nombre] -> Nombre
;;;  (horner x liste-coeffs) rend l'évaluation, pour la valeur x, du
;;;  polynôme a0 + a1.x + a2.x^2 + ... + an.x^n, si liste-coeffs est
;;;  la liste de ses coefficients '(a0 a1 ... an).
```

.

Question 1

Donner une autre *définition* de la fonction `horner` nommée `horner-reduce` qui utilise la fonction `reduce`. On rappelle la spécification de fonction `reduce`

```
;;; reduce : (alpha * beta -> beta) * beta * LISTE[alpha] -> beta
;;; (reduce f fin liste)  renvoie la valeur (f e1 (f e2 ... (f eN fin)..))
;;;  si la liste comporte les termes e1 e2  ... eN
```

Exercice 66. Gonfler une liste

Cet exercice utilise la fonction `reduce` (voir exercices **??** page **??**) pour répondre à une des questions de l'exercice **??** page **??**.

Question 1

Écrire la *signature* et une *définition* de la fonction `gonflebis`, en utilisant la fonctionnelle `reduce`, qui étant donnée une liste L, rend la liste où chaque élément x de L est remplacé par un nombre aléatoire d'occurences de x. Par exemple :

```
(gonflebis(list 1 2 1 4))
→(1 1 2 2 2 2 2 2 1 1 1 1 4 4 4 4 4 4 4 4)
(gonflebis(list 1 2 1 4))
→(1 1 1 1 1 2 2 2 1 1 1 1 1 1 4 4 4 4 4)
```

On utilisera la fonction `(random k)`, qui étant donné un entier naturel k rend un nombre entier aléatoire entre 0 et $k-1$.

Chapitre 7

Problèmes

Exercice 67. Y-a-t-il un point dans l'intervalle ?

La donnée de cet exercice est un ensemble de points et un intervalle de l'axe réel, et le but de l'exercice est de rechercher un point qui appartienne à l'intervalle.

Ici l'ensemble des points est représenté par une liste, où l'ensemble des points est représenté par un arbre).

Un point de l'axe réel est déterminé par son abscisse x, et un intervalle est déterminé par les abscisses de ses extrémités. Un intervalle sera donc représenté par une liste de deux nombres : `Intervalle = COUPLE[Nombre Nombre]`.

On utilisera les deux accesseurs suivants :

```
;;; deb : Intervalle -> Nombre
;;; (deb iv) renvoie l'abscisse de l'extrémité gauche de l'intervalle iv

;;; fin : Intervalle -> Nombre
;;; (fin iv) renvoie l'abscisse de l'extrémité droite de l'intervalle iv
```

Question 1

Écrire une définition des deux accesseurs `deb` et `fin`.

Question 2

Écrire une définition du prédicat `est-dans-interv?`, qui vérifie si un point x appartient à un intervalle iv. Par exemple :

`(est-dans-interv? 5 '(8 12))` → `#f`
`(est-dans-interv? 9 '(8 12))` → `#t`

Question 3

On suppose que l'ensemble des points est représenté par une liste (l'ordre des points dans la liste est quelconque). Écrire une définition du semi-prédicat `LN-recherche` répondant à la spécification suivante :

```
;;; LN-recherche :  LISTE[Nombre]*Intervalle -> Nombre + #f
;;; (LN-recherche L iv) renvoie le premier point de L qui est dans l'intervalle iv,
;;; si un tel point existe ; renvoie #f si L ne contient pas de point dans iv.
```

Par exemple :

`(LN-recherche '(2 4 12 6 10 5 9) '(5 8))` → `6`
`(LN-recherche '(2 4 12 6 10 5 9)  '(13 31))` → `#f`

Question 4

Le programme suivant est censé répondre à la même spécification que précédemment, mais il provoque une erreur.

```
(define (LN-rechercheFAUX L iv)
  (define (LN-rech-x1-x2 L)
    (if (pair? L)
        (let ((val (car L)))
          (if (and (<= x1 val) (<= val x2))
              val
              (LN-rech-x1-x2 (cdr L))))
        #f))
  (let ((x1 (deb iv))
        (x2 (fin iv)))
    (LN-rech-x1-x2 L)))
```

Par exemple :

```
(LN-rechercheFAUX '(2 12 6 5 9) '(5 8))
  -> reference to undefined identifier: x1
```

Rechercher pourquoi la fonction `LN-rechercheFAUX` n'est pas correcte, et la corriger en une fonction `LN-rechercheJUSTE`.

Question 5

On suppose à présent que les points de la liste sont triés en ordre croissant de leurs abscisses. Écrire une définition du semi-prédicat `LTN-recherche` répondant à la spécification suivante :

```
;;; LTN-recherche : LISTE[Nombre]*Intervalle -> Nombre + #f
;;; (LTN L iv) renvoie ... (voir question 3)
;;; HYPOTHESE : la liste L est triée
```

Remarque : Lorsque la liste ne vérifie pas l'hypothèse, on ne peut rien assurer sur le résultat de la fonction. La spécification *n'engage le programmeur* que si son programme est *utilisé sous les hypothèses requises*.

Par exemple avec la fonction donnée en solution, lorsque la liste n'est pas triée le résultat obtenu peut être *faux* (dans le premier exemple ci-dessous, 7 est dans l'intervalle et la fonction renvoie #f), *juste* (deuxième exemple) ou *à moitié juste* (dans le troisième exemple, la fonction renvoie 10, alors que le plus petit point dans l'intervalle est 9).

`(LTN-recherche '(2 3 12 20 7 25) '(5 8))` → `#f`
`(LTN-recherche '(1 2 9 6 3 10 15) '(8 12))` → `9`
`(LTN-recherche '(1 2 10 9 15 20) '(8 12))` → `10`

Exercice 68. Y-a-t-il un point dans le rectangle ?

On traite ici le même problème que dans l'exercice **??** page **??**, mais les points sont situés dans le plan, et l'on recherche s'il y a en un à l'intérieur d'un certain rectangle.

L'ensemble des points est représenté par une liste de coordonnées `(x y)`, et le rectangle est déterminé par deux points diagonaux.

Question 1

Dans cette question, on demande de définir quelques fonctions utilitaires pour manipuler les points et les rectangles.

Un point est représenté par une liste de deux nombres `(x y)`, son abscisse et son ordonnée. Un rectangle est représenté par les deux points diagonaux de sa première diagonale : `(x1 y1)` et `(x2 y2)`, avec `x1` $\leq$ `x2` et `y1` $\leq$ `y2`. Dans tout ce qui suit on supposera que le rectangle est bien formé, c'est-à-dire que ses points diagonaux vérifient bien les hypothèses précédentes. Le point `(x1 y1)` est appelé *extrémité inférieure* du rectangle, et `(x2 y2)` est appelé *extrémité supérieure*.

On utilisera les types

– `Point = COUPLE[Nombre Nombre]`, et

– `Rectangle = COUPLE[Point Point]`, avec la condition `x1` $\leq$ `x2` et `y1` $\leq$ `y2`.

Donner les définitions des fonctions suivantes :

```
;;; absc : Point -> Nombre
;;; (absc P) renvoie l'abscisse de P

;;; ordo : Point -> Nombre
;;; (ordo P) renvoie l'ordonnée de P

;;; basGauche : Rectangle -> Point
;;; (basGauche Rect) renvoie l'extrémité inférieure de Rect

;;; hautDroit : Rectangle -> Point
;;; (hautDroit Rect) renvoie l'extrémité supérieure de Rect
```

Question 2

Écrire la *signature* et une définition de la fonction `(estDansRect? P Rect)` qui renvoie `#t` si et seulement si le point `P` appartient, au sens large, au rectangle `Rect`. Par exemple :

`(estDansRect? '(3 5) '((2 3) (5 10)))` $\rightarrow$ `#t`

Question 3

L'ensemble des points est représenté par une liste. Écrire la *signature* et une définition du semi-prédicat `rechercheListeP` qui recherche s'il existe un point de la liste qui est à l'intérieur d'un rectangle donné.

```
;;; rechercheListeP : Rectangle * LISTE[Point] -> Point + #f
;;; (rechercheListeP Rect LPoints) renvoie un point de LPoints (le plus
;;; prés du début de la liste) qui est dans le rectangle Rect
;;; et renvoie #f si la liste LPoints ne contient pas de point dans Rect
```

Par exemple :

```
(rechercheListeP
   '((0 0) (5 5))
   '((-1 -1) (-1 0) (2 2) (2 1) (6 5)))

   →(2 2)
```

Question 4

On suppose à présent que les points de la liste sont triés en ordre croissant de leurs abscisses (mais sont en ordre quelconque pour les ordonnées).

Écrire le semi-prédicat `rechercheListePTrieeAbsc` répondant à la spécification suivante :

```
;;; rechercheListePTrieeAbsc : Rectangle * LISTE[Point] -> Point + #f
;;; (rechercheListePTrieeAbsc Rect LPoints) renvoie un point de LPoints
;;; qui est dans le rectangle Rect et renvoie #f si la liste LPoints ne
;;; contient pas de point dans Rect.
```

Par exemple :

```
(rechercheListePTrieeAbsc
   '((0 0) (5 5))
   '((-1 -1) (-1 0) (2 2) (2 1) (6 5)))
```

→`(2 2)`

Lorsque la liste ne vérifie pas l'hypothèse, on ne peut rien assurer sur le résultat de la fonction. Tester le comportement de votre programme dans des situations analogues à celles évoquéees dans l'exercice **??** page **??**.

Chapitre 8

Préambule des contrôles

Voici le préambule habituel des devoirs sur table et des examens, prenez le temps de le lire avant le premier contrôle :

« Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Scheme.

Le sujet comporte ... pages. Ne pas désagrafer les feuilles.

Répondre sur la feuille même, dans les boîtes appropriées. La taille des boîtes suggère le nombre de lignes de la réponse attendue. Le barème apparaissant dans chaque boîte n'est donné qu'à titre indicatif. Le barème total est lui aussi donné à titre indicatif : ... points.

La clarté des réponses et la présentation des programmes seront appréciées. Les questions peuvent être résolues de façon indépendante. Il est possible, voire même utile, pour répondre à une question, d'utiliser les fonctions qui sont l'objet des questions précédentes.

Pour vous faire gagner du temps, il ne vous est pas systématiquement demandé, en plus de la définition, la spécification entière d'une fonction. Bien lire ce qui est demandé : seulement la définition ? la signature et la définition ? la spécification et la définition ? seulement la spécification ?

Lorsque la signature d'une fonction vous est demandée, vous veillerez à bien préciser, avec la signature, les éventuelles hypothèses sur les valeurs des arguments. *»*

En effet, dans une situation hors contrôle « écrire une fonction » signifie, sans aucune exception, écrire sa signature, sa spécification, les hypothèses que doivent respecter les paramètres, le jeu de tests le plus complet possible pour valider la fonction et bien sûr sa définition. Tout ceci afin d'assurer une bonne utilisation des fonctions écrites.

En situation de contrôle, le temps étant limité, nous devons choisir ce qui doit être contrôlé et c'est pour cette raison que nous précisons ce qui vous est demandé. Ainsi, par exemple, nous vous demandons rarement la spécification d'une fonction si nous l'avons donnée dans l'énoncé...

En conclusion soyez attentif à ce qui vous est demandé.