Exercices « Processus d'évaluation »
Première saison – Lot 2
UPMC DEUG MIAS
Revision: 1.1

Anne Brygoo, Titou Durand, Pascal Manoury,
Maryse Pelletier, Christian Queinnec, Michèle Soria
Université Paris 6 — Pierre et Marie Curie

septembre 2002 — janvier 2003

Ce deuxième lot d'exercices fait suite aux exercices élémentaires donnés précédemment. Il commence par deux exercices graphiques simples, et la suite est divisée en six parties, de difficulté croissante :

1. Des exercices de récursion sur les nombres :
   – schéma de récursion linéaire,
   – évaluation par substitution,
   – schéma de récursion dichotomique.
2. Des exercices simples sur les listes :
   – manipulation des opérations de base,
   – fonctions dont les paramètres sont des nombres, et qui rendent une liste.
3. Des exercices de récursion sur les listes :
   – schéma de récursion linéaire,
   – fonctions sur les listes, qui rendent une valeur atomique,
   – fonctions sur les listes, qui rendent une liste,
   – utilisation de `map` et de `filtre`,
   – listes d'associations,
   – récursion croisée,
   – récursion double,
   – utilisation de `reduce`,
   – utilisation de la citation.
4. Des exercices sur les chaînes de caractères, et sur les types `Ligne` et `Paragraphe`, qui permettent de faire des affichages variés :
   – utilisation des fonctions sur les chaînes,
   – manipulation des types `Ligne` et `Paragraphe`.
5. Une introduction à la notion de barrière d'abstraction,
6. Des exercices longs mettant en œuvre toutes les notions vues précédemment.

1 Exercices graphiques simples

Débuter par l'un des deux exercices graphiques suivants (à faire en TP, après l'avoir préparé en TD) : « Dessin d'un sablier » (page 1) ou « Dessin d'une tour » (page 2).

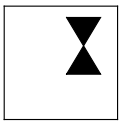
Exercice 1 – Dessin d'un sablier

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 et l'on doit donc pouvoir appeler une telle fonction avec n'importe quelles valeurs, sous réserve 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 des dessins comme dessiné ci-contre.

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.



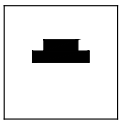
Exercice 2 – Dessin d'une tour

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 des dessins comme ci-contre.

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



2 Récursion sur les nombres

On étudie dans cette section le schéma de récursion linéaire sur les nombres et le mécanisme d'évaluation par substitution.

1. Faire les exercices « Quelle est cette fonction f » (page 2), « Quelle est cette fonction g » (page 3), « Somme des n premiers impairs » (page 3),
2. Puis l'exercice « Nombres et chiffres » (page 3),
3. Enfin l'exercice « Calcul du pgcd » (page 4), pour étudier des schémas de récursion plus complexes, et travailler sur différents algorithmes pour résoudre un même problème. (Cet exercice pourra être traité plus tard.)
4. Les exercices faisant intervenir du graphique pourront être préparés en TD, et fait en TP. On pourra travailler sur les pyramides, en commençant par « Dessin de pyramide » (page 5) ou sa version plus directive « Pyramides » (page 6). Les autres exercices sont facultatifs.

Exercice 3 – Quelle est cette fonction f

Une récursion simple sur les nombres

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. Donner la spécification de cette fonction.

Exercice 4 – Quelle est cette fonction g

Une récursion simple sur les nombres : fonction avec 2 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. Donner la spécification de cette fonction.

Question 2 : Dans la fonction `g`, le paramètre `n` est inchangé lors des appels récursifs. On peut donc améliorer le programme en définissant une fonction interne (récursive) à un seul argument, dans laquelle `n` est global.

Écrire une définition de la fonction `somme-carres` , ayant la même spécification que la fonction `g`, qui utilise une telle fonction auxiliaire.

Exercice 5 – Somme des n premiers impairs

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

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

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

Exercice 6 – Nombres et chiffres

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 définition de la fonction `somme-des-chiffres` qui rend la somme des chiffres d'un entier positif *n*. Par exemple

```
(somme-des-chiffres 546)  → 15
(somme-des-chiffres 7)  → 7
```

Question 2 :

Écrire la définition de la fonction `nombre-de-chiffres` qui rend le nombre de chiffres significatifs d'un entier naturel $n > 0$. Par exemple

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

Question 3 :

Écrire la définition du prédicat `existe-chiffre?` qui, étant donné un chiffre *c* entre 0 et 9, et un entier naturel $n > 0$, 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 6045)  → #T
```

Question 4 :

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

Question 5 :

Écrire la 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 7 – Calcul du pgcd

Implantation de fonctions de calcul du *plus grand diviseur commun* (*pgcd*) de deux nombres selon différents algorithmes.

Question 1 :

Un premier algorithme consiste à calculer le *pgcd* de deux nombres 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 une définition de la fonction `pgcd` qui implante cet algorithme.

Question 2 :

Le deuxième algorithme proposé procède par divisions successives. Il utilise les égalités suivantes :

– $pgcd(m,0) = m$
– $pgcd(m,n) = pgcd(n, m \bmod n)$, 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 8 – 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 la fonction récursive `quotient-et-reste` de signature

`;;;` *quotient-et-reste: nat * nat/>0/ -> NUPLET[nat nat]*

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. à l'étape de base, lorsque $m < n$, le quotient est 0 et le reste est m ;
2. à l'étape récursive, lorsque $m \geq n$, 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.

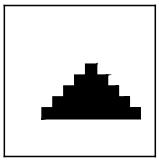
Question 2 :

Écrire une fonction qui vérifie que les résultats que vous obtenez avec votre fonction sont les mêmes que ceux que l'on obtient avec les fonctions `quotient` et `remainder` de Scheme.

Indications : si `(quotient-et-reste m n)` renvoie le couple `(q r)` alors on doit avoir que `(quotient m n)` est égal à `q` et `(remainder m n)` est égal à `r`.

Exercice 9 – Dessin de pyramide

Le but de cet exercice est de dessiner des pyramides, avec un nombre quelconque d'étages.
Sur le dessin ci-contre, par exemple, on a dessiné une pyramide à 5 étages.
Dans cet exercice on ne donne aucune indication sur les paramètres nécessaires pour faire un tel dessin, et plusieurs paramétrages sont possibles.
Pour travailler avec des paramètres précis, faire l'exercice « Pyramides » (page 6)



Question 1 :

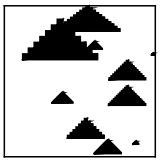
Écrire une fonction (récursive) qui dessine une pyramide.

Modifier cette fonction pour qu'elle signale une erreur si la pyramide ne tient pas (en hauteur ou en largeur) dans le cadre de dessin de DrScheme.

Écrire une fonction qui dessine une pyramide *tenant toujours* à l'intérieur du cadre de dessin de DrScheme.

Exercice 10 – Champ de pyramides

Le but de cet exercice est de dessiner un champ de pyramides, comme ci-contre. Afin que les pyramides aient des tailles différentes et soient placés à des endroits différents, il faut pouvoir dessiner une pyramide avec des dimensions et un placement aléatoire.



Question 1 : Écrire une fonction `pyramide-aleatoire` de spécification :

```
;;; pyramide-aleatoire:  -> Image
;;; (pyramide-aleatoire) renvoie une pyramide, construite à partir de
;;; paramètres choisis aléatoirement
```

Quelques indications : Il existe en Scheme une fonction `random` avec la spécification suivante :

```
;;; random : nat -> nat
;;; (random n-max) retourne un entier aléatoire compris entre 0 (inclus) et n-max (exclu),
;;; n-max étant un entier inférieur strictement à 2^31.
```

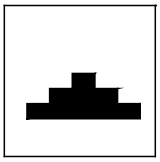
Pour tirer un nombre rationnel aléatoire entre 2 bornes *min* et *max*, on peut discrétiser l'intervalle $[min, max]$ en un grand nombre N de points (par exemple $N = 100000$), puis calculer

$$min + (max - min)\ \frac{random(N)}{N}\ .$$

Question 2 : Écrire une fonction, nommée `champ`, qui dessine un champ de p pyramides aléatoires.

Exercice 11 – 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 ?

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 une fonction `rectangle` de paramètres `x, y, l, h` qui dessine 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 une fonction récursive `pyramide1` de paramètres `n, x, y, l, h` qui dessine 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.

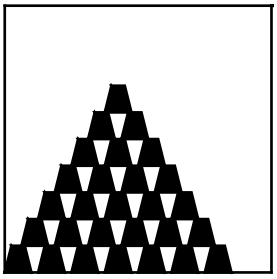
Question 3 : Écrire une fonction `pyramide2` de paramètres `n, x, y, l, h` qui dessine une pyramide en vérifiant qu'elle tient dans le cadre de dessin de DrScheme, et signale une erreur si ce n'est pas le cas. Cette fonction indiquera la (ou les) dimension(s) trop grande(s) dans le message d'erreur.

Question 4 : Écrire une fonction `pyramide3` de paramètres `n, x, y, l, h` qui dessine 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 12 – 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 une 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 une fonction `pyramide-gobelet1` de paramètres `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 une 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).

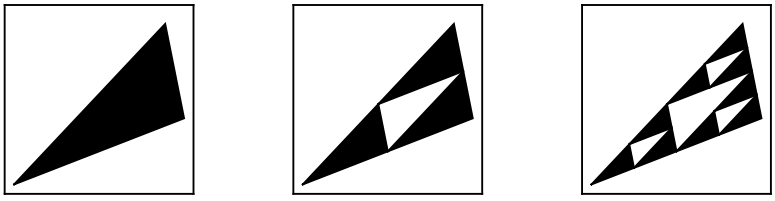
Question 4 : Écrire une fonction `pyramide-gobelet3` de paramètres `x, y, h, nb-gob` qui dessine une pyramide construite avec au plus $nb-gob$ gobelets, de coin bas-gauche (x,y), h étant la hauteur d'un gobelet. La pyramide *doit tenir* à l'intérieur du cadre de dessin de DrScheme.

Exercice 13 – Triangles de Sierpinski

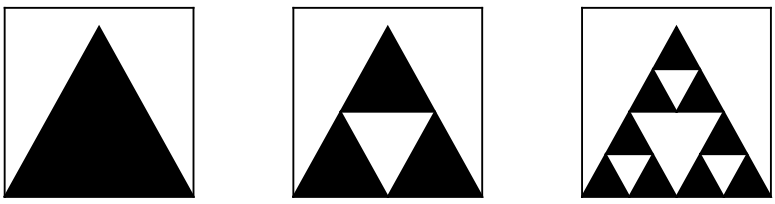
Voici une définition récursive des *triangles de Sierpinski* :

- 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 une fonction récursive `sierpinski` qui dessine un triangle de Sierpinski.

Pensez à utiliser la fonction `resize-image` pour mieux voir les petits triangles.

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 une 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 une fonction `sierpinski-equilateral` qui répond à ce problème.

3 Exercices simples sur les listes

Ces exercices ont pour objectif de faire manipuler les opérations de base sur les listes, et de construire des listes. On pourra traiter

1. L'exercice « Liste de longueur au moins 3 » (page 9), et éventuellement l'exercice « Liste des racines d'une équation du second degré » (page 9),
2. Un exercice parmi « Liste de répétitions » (page 10), et « Intervalle d'entiers » (page 10).

Exercice 14 – Liste des racines d'une équation du second degré

Calcul des racines d'une équation du second degré.

Question 1 :

En utilisant des alternatives, écrire 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 15 – Liste de longueur au moins 3

Pour vérifier qu'une liste comporte au moins trois éléments, on pourrait penser utiliser la primitive `length` 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)
  (>= (length L) 3))
```

Cette solution n'est pas efficace, car la primitive `length` 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.

Question 1 :

Écrire la définition d'un prédicat `longueur-sup3` qui, étant donné une liste L, vérifie que la liste a au moins trois éléments, sans parcourir toute la liste.

Question 2 :

Écrire la définition d'un prédicat `longueur-egale3` qui, étant donné une liste *L*, vérifie que la liste a exactement trois éléments.

Exercice 16 – 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 prédéfinie `length` et d'écrire :

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

Écrire la définition d'un prédicat `lg?` qui prend un entier naturel et une liste et vérifie si la liste a cet entier pour taille. Ainsi :

```
(lg? 0 (list))  → #T
(lg? 2 (list "a" "bb" "ccc"))  → #F
(lg? 2 (list "a" "bb"))  → #T
```

Question 2 :

Regardez la définition suivante et écrivez, en utilisant `verifier` une expression de test montrant que la fonction `lg-fausse?` est bien fausse.

```
;;; lg-fausse?: Naturel * LISTE[a] -> bool
;;; (lg-fausse? n L) vérifie que la liste L a pour longueur n.
(define (lg-fausse? n L)
  (if (> n 0)
      (lg-fausse? (- n 1) (cdr L))
      (not (pair? L)) ) )
```

Exercice 17 – Liste de répétitions

Petite fonction simple de construction de liste.

Question 1 :

Écrire la définition de la fonction `repete` qui étant donnés un naturel *n* et un nombre *x* construit la liste contenant *n* occurences de *x*. Par exemple

```
(repete 0 3)  → ()
(repete 3 0)  → (0 0 0)
```

Exercice 18 – Intervalle d'entiers

Autre petite fonction de construction de liste simple.

Question 1 :

Écrire la définition de la fonction `intervalle` qui étant donnés deux nombres entiers *n* et *m* construit la liste contenant tous les entiers de *n* inclus à *m* inclus. On suppose que *n* est inférieur ou égal à *m*. 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 $[n,m]$ est vide lorsque *m* est strictement inférieur à *n*. Écrire une fonction `intervalle-gen` telle que `(intervalle-gen n m)` renvoie la liste des entiers compris entre *n* inclus et *m* inclus, sans faire l'hypothèse que $n < m$. Par exemple,

```
(intervalle-gen -2 2)  → (-2 -1 0 1 2)
(intervalle-gen 2 -2)  → ()
```

4 Récursion sur les listes

Cette section est divisée en trois parties, en ordre croissant de difficulté.

4.1 Récursion linéaire

Tous les exercices présentés ici sont des parcours de listes qui produisent une valeur atomique (nombre ou booléen). On pourra traiter

1. Les exercices « Maximum d'une liste de nombres » (page 11) et « Présence d'un élément dans une liste » (page 11),
2. Puis les exercices « Sommes sur les éléments d'une liste » (page 12), « Nombre d'occurrences » (page 12) et éventuellement l'exercice « Nombre de maximums d'une liste » (page 12),
3. Enfin les exercices « Recherche du n-ième élément d'une liste » (page 13) et « Moyenne de trois nombres » (page 13), et « Schéma de Horner » (page 14) (ce dernier propose aussi une solution utilisant reduce).

Exercice 19 – Maximum d'une liste de nombres

le but de cet exercice est de calculer le maximum d'une liste de nombres.

Question 1 : Écrire 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
```

Exercice 20 – Présence d'un élément dans une liste

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

Question 1 :

Écrire une définition du prédicat `est-dans?` qui, étant donné un élément *e* et une liste *L* rend vrai si et seulement si *e* est un élément de *L*. On utilisera ici des alternatives.

```
(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 `or` et `and`, écrire une définition de la fonction `est-dans-bis?`, qui répond à la même spécification que la fonction `est-dans?`.

Exercice 21 – 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 et la somme des carrés des éléments.

Question 1 :

Donner 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 d'une liste vide est 0. Par exemple

```
(somme-liste (list 2 5 7 3))  → 17
(somme-liste (list))  → 0
```

Question 2 : Donner 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 :

Après avoir écrit une fonction `carre`, qui rend le carré d'un nombre, utiliser la fonction `map` et la fonction définie à la première question pour écrire une définition de la fonction `somme-liste-carres-bis` qui a la même spécification que la fonction `somme-liste-carres`.

Exercice 22 – 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 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 23 – Nombre de maximums d'une liste

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

Question 1 : Écrire 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.

```
(nombre-de-max (list 1 8 2 8 6 5 8))  → 3
```

Dans cette question on utilisera les fonctions `nombre-occurrences` et `max-liste` définies dans les exercices « Nombre d'occurrences d'un élément dans une liste » (page 12) et « Maximum d'une liste de nombres » (page 11).

Question 2 : La solution de la question précédente est peu efficace, au sens où l'on parcourt deux fois la liste : une fois pour évaluer le maximum, et une autre fois pour calculer son nombre d'occurrences.

Pour résoudre le problème en un seul parcours de la liste, on va définir une fonction `qui-combien`, qui étant donné une liste non vide de nombres rend le doublet formé du maximum de la liste et de son nombre d'occurrences :

```
;;; qui-combien :  LISTE[Nombre]/non vide/ -> NUPLET[Nombre  nat]
;;; (qui-combien L) rend le doublet formé du maximum de L et de son nombre d'occurrences
(qui-combien (list 1 8 2 8 6 5 8))  → (8 3)
```

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

Question 3 : Écrire la fonction `nombre-de-max-bis` qui fait appel à `qui-combien` pour calculer le nombre d'occurrences du maximum dans une liste de nombres.

Exercice 24 – 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 :

Écrire une fonction `n-ieme` qui, étant donnés un entier *n* et une liste, renvoie le *n*-ième élément de la liste (les éléments d'une liste étant numérotés à partir de 0). On renverra un message d'erreur lorsque le traitement est impossible.

```
(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)) →  n-ieme: ERREUR: position trop grande
(n-ieme -1 (list 8 2)) →  n-ieme: ERREUR: position doit être >=0
```

Exercice 25 – Moyenne d'une liste de nombres

Question 1 : Écrire la définition de la fonction `somme-liste` qui, étant donnée une liste de nombres, retourne la somme des éléments de la liste. Cette fonction rendra 0 lorsque la liste est vide. Par exemple :

```
(somme-liste (list 4 12 10))  → 26
(somme-liste (list))  → 0
```

Question 2 : En déduire 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)) → moyenne-liste ERREUR: liste vide
```

Question 3 : Le calcul de la moyenne dans la question prédédente manque d'efficacité. En effet, dans le cas d'une liste non vide, on parcourt deux fois la liste : une fois pour faire la somme, et une fois pour

calculer la longueur (même en utilisant la primitive `length`. On peut ne parcourir qu'une seule fois la liste, en définissant une fonction `somme-longueur-liste` qui calcule en même temps sa somme et sa longueur. Écrire la définition de cette fonction, qui a pour spécification

```
;;; somme-longueur-liste : LISTE[Nombre] -> NUPLET[Nombre Nombre]
;;; (somme-longueur-liste L) rend le doublet 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)
```

Question 4 : Écrire une définition de la fonction `moyenne-liste-bis`, de même spécification que `moyenne-liste`, mais qui fait appel à la fonction `somme-longueur-liste`.

Exercice 26 – Schéma de Horner

Étant donné un polynôme $a_0 + a_1x + a_2x^2 + ... + a_nx^n$, 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 :

$$a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 = a_0 + x*(a_1 + x*(a_2 + x*(a_3 + x*a_4)))$$

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 « Fonctions polynomiales » (page **??**).

Question 1 :

Écrire 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
```

Question 2 :

Écrire une autre fonction `horner-reduce` qui définit le calcul de Hörner en utilisant 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
```

4.2 Construction de listes

Tous les exercices présentés ici sont des parcours de listes qui produisent des listes. C'est l'occasion de présenter les itérateurs `map` et `filtre`. On pourra traiter

1. Les exercices « Liste des carrés d'une liste » (page 14), « Liste croissante » (page 15), « Begaie — debegaie » (page 15)
2. Un exercice parmi « Liste des éléments plus grands qu'un nombre donné » (page 16) et « Enlever toutes les occurrences d'un élément » (page 16).
3. L'exercice « Enlever les doublons d'une liste croissante » (page 16), qui utilise une récursion croisée.

Exercice 27 – 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 une fonction *récursive* `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)
```

Question 2 :

Écrire une fonction *non récursive* `liste-carres-bis`, de même spécification que `liste-carres`, qui itère la fonction `carre` : $x \rightarrow x^2$, sur une liste de nombres.

Exercice 28 – Liste croissante

Le but de cet exercice est de tester si une liste de nombres est croissante.

On dit qu'une liste `(e1 e2 ... en)` est croissante (au sens large) lorsque $e1 \leq e2 \leq ... \leq en$.

Question 1 :

Écrire un prédicat récursif `croissante?` qui teste si une liste de nombres est croissante au sens large. Notons que la liste vide et les listes réduite à un seul nombre sont croissantes.

```
(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 29 – 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 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 une définition de la fonction `debegaie` qui, étant donnée une liste bégayée, restitue la liste initiale. C'esr-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 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 : la liste donnée n'est pas « bégayée »*

Exercice 30 – 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 « Liste des éléments plus grands qu'un nombre donné » (page 16)

Question 1 :

Écrire 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")
```

Question 2 :

Traiter le même problème qu'à la question précédente, en définissant une fonction `moins-occurrences-bis` en utilisant la fonction `filtre` dont on rappelle la spécification :

*;;; filtre : (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]*

;;; (filtre pred L) renvoie la liste des éléments de L qui satisfont le prédicat pred

Indication : pour pouvoir utiliser `filtre`, 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*.

Exercice 31 – 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 « Enlever toutes les occurrences d'un élément » (page 16)

Question 1 : Écrire une fonction récursive `plus-grands` qui, étant donnés un nombre *e* et une liste de nombre *L*, renvoie la liste des éléments de *L* supérieurs ou égaux à *e*.

```
(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))  → ()
```

Question 2 : On va traiter le même problème qu'à la question précédente, en définissant une fonction `plus-grands-bis` utilisant la fonction `filtre` dont on rappelle la spécification :

*;;; filtre : (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]*

;;; (filtre pred L) renvoie la liste des éléments de L qui satisfont le prédicat pred

Pour pouvoir utiliser `filtre`, il faut définir un prédicat, interne à la fonction `plus-grands-bis`, qui étant donné un nombre *x*, renvoie vrai ssi $x >= e$.

Exercice 32 – Enlever les doublons d'une liste croissante

Le but de cet exercice est de supprimer les doublons d'une liste croissante. C'est l'occasion de traiter une récursion croisée.

Question 1 :

Écrire une fonction `moins-doublons-prefixe` qui supprime tous les éléments situés en début d'une liste égaux à une certain *e* (et seulement ceux-là). Par exemple

```
(moins-doublons-prefixe 1 (list 1 1 2 2 3 3 3))  → (2 2 3 3 3)

(moins-doublons-prefixe 1 (list 1 1 1))  → ()

(moins-doublons-prefixe 1 (list 2 2 3 3 3))  → (2 2 3 3 3)
```

Question 2 :

Écrire une définition de la fonction `moins-doublons` qui, à partir d'une liste de nombres croissante au sens large (qui comporte donc éventuellement plusieurs occurrences de chaque élément), rend la liste croissante au sens strict où chaque élément n'apparaît qu'une seule fois. Par exemple

```
(moins-doublons (list 1 2 2 3 4 4 4))  → (1 2 3 4)
```

Indication : l'hypothèse de croissance de la liste nous garantit que tous les éventuels doublons apparaissent consécutivement dans la liste. Par (contre) exemple, la liste `(list 1 2 2 3 1 4 4 4)` n'est pas croissante. On peut donc utiliser la fonction `moins-doublons-prefixe`.

Question 3 :

Dans la question précédente, l'imbrication des appels à `moins-doublons` et `moins-doublons-prefixe` a pour effet que cette seconde fonction "rend la main" à `moins-doublons` aprés avoir débarassé son argument des éventuels doublons. On pourrait imaginer qu'elle continue plutôt le travail en appelant elle-même la fonction principale `moins-doublons`. On obtiendrait ainsi deux fonctions *mutuellement récursives*.

Écrire deux fonctions mutuellement récursives `moins-doublons-rec` et `moins-doublons-prefixe-rec` telles que `moins-doublons-rec` supprime les doublons d'une liste croissante de nombres.

4.3 Exercices plus compliqués sur les listes

Ces exercices sont un peu plus complexes. Le premier introduit des listes d'association, le deuxième une récursion double, et les suivant travaillent avec l'itérateur `reduce` On pourra traiter dans l'ordre

1. L'exercice « Fréquences » (page 17),
2. L'exercice « Tri fusion » (page 18),
3. L'exercice « Exercices avec reduce » (page 19),
4. Et enfin éventuellement l'exercice « Retour sur la croissance d'une liste » (page **??**), et l'exercice « Gonfler et dégonfler une liste » (page 20), qui est une généralisation des exercices « Begaie — debegaie » (page 15) et « Enlever les doublons d'une liste croissante » (page 16).

Exercice 33 – 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'association dans laquelle chaque élément de la liste est associé à son nombre d'occurrences dans la liste.

Question 1 :

Écrire 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 :

En déduire la 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 « Nombre d'occurrences d'un élément dans une liste » (page 12).

On demande ici de le faire en utilisant la fonction `frequence`. Il s'agit donc d'écrire la définition de la fonction `nbre-occ` qui étant donné 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
```

Penser à utiliser la primitive `assoc`.

Question 4 :

Écrire 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? 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 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 34 – 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 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 des 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 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)
```

Exercice 35 – Exercices avec reduce

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

La fonction `reduce` a pour spécification :

```
;;; 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
```

Question 1 :

En utilisant la fonction `reduce`, donner une définition pour les 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 « Sommes sur les éléments d'une liste » (page 12).

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.

Voir aussi exercice « Maximum d'une liste de nombres » (page 11).

Question 3 :

En utilisant la fonction `reduce`, donner une définition de la fonction `nombre-occurrences`, qui rend le nombre d'occurences d'un élément donné dans une liste.

Voir aussi exercice « Nombre d'occurrences d'un élément dans une liste » (page 12).

Exercice 36 – 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 « Liste croissante » (page 15).

Question 1 :

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

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

Par exemple

```
(inf 2 2)  → 2
(inf 2 3)  → 2
(inf 2 1)  → #F
(inf 2 #F)  → #F
```

Question 2 :

En utilisant les fonctions `reduce` et `inf`, donner une définition de la fonction `croissante-reduce?` telle que `(croissante-reduce? L)` renvoie le booléen `#t` si `L` est une liste de nombres est croissante au sens large ; et le booléen `#f` sinon. On fait l'hypothèse que tous les nombres de la liste sont inférieurs à 1000.

Question 3 :

L'utilisation de `reduce` nous a obligé à faire l'hypothèse que les éléments de la liste étaient tous inférieurs à une certaine borne. On veut maitenant une solution plus générale.

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

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

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

Voici quelques exemples de d'applications

```
(inf2 4 4)  → 4
(inf2 4 5)  → 5
(inf2 4 3)  → #F
(inf2 #F 5)  → #F
```

Question 6 :

Écrire une fonction `croissante-accumulee?`, *récursive terminale*, qui teste si une liste de nombres est croissante au sens large. On utilisera une fonction interne auxiliaire pour accumuler les résultats.

Exercice 37 – Gonfler et dégonfler une liste

Cet exercice utilise la fonction `reduce` (voir exercice « Exercices avec reduce » (page 19)) pour généraliser l'exercice « Begaie —debegaie » (page 15). La troisième question fait appel à la notion d'accumulateur.

Question 1 :

Écrire une définition de la fonction `gonfle` 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. Voici par exemple deux exécutions de `gonfle`

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

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

Question 2 :

Après avoir "gonflé" la liste, on veut la "dégonfler". Écrire 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(gonfle 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 une fonction `degonfle-renverse`, *récursive terminale* 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 4 4))` → `(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 pour accumuler les résultats.

5 Ligne et Paragraphe

Exercices sur les types Ligne et Paragraphe

Exercice 38 – Conjugaison

Cet exercice fait travailler la synthèse de lignes et paragraphes.

Question 1 :

Dans cette question, il faudrait que vous vous démerdassiez à écrire la fonction, nommée `subjonctif-imparfait-demerder`, calculant la conjugaison du verbe « démerder » à l'imparfait du subjonctif. Ainsi

```
 (subjonctif-imparfait-demerder)  → "
que je démerdasse
que tu démerdasses
qu'il démerdât
que nous démerdassions
que vous démerdassiez
qu'elles démerdassent
"
```

Question 2 :

Généraliser la fonction précédente à tout verbe du premier groupe (les verbes qui se terminent par « er »). La fonction à écrire se nommera `subjonctif-imparfait`, prendra en unique argument l'infinitif

du verbe à conjuguer (sous forme d'une chaîne de caractères) et calculera la conjugaison de ce verbe au subjonctif imparfait. Ainsi,

```
 (subjonctif-imparfait "épater")  → "
que j'épatasse
que tu épatasses
qu'il épatât
que nous épatassions
que vous épatassiez
qu'elles épatassent
"
```

Dans un premier temps, on ne se souciera pas des élisions. Dans un dernier temps, on se dispensera du traitement des exceptions comme les « h » aspirés.

Exercice 39 – Paragraphes et frises

Le but de cet exercice est de faire manipuler des chaînes de caractères, et les types `Ligne` et `Paragraphe`. On rappelle qu'une `Ligne` est une chaînes de caractères ne comportant pas de caractère de fin de ligne, et qu'un `Paragraphe` est une chaîne de caractères débutant par un caractère de fin de ligne, et suivi d'une suite de lignes, terminées chacune par un caractère de fin de ligne.

Question 1 : Écrire une fonction, nommée `colle-horizontale`, qui, étant donnés une ligne `L` et un entier `n`, rend la ligne formée de `n` fois la ligne `L`. Par exemple

```
 (colle-horizontale "bon" 2)  → "bonbon"
```

Question 2 : Écrire une fonction, nommée `colle-verticale`, qui, étant donnés une ligne `L` et un entier `n`, rend le paragraphe formé de `n` fois la ligne `L`. Par exemple

```
 (colle-verticale "bon" 2)  → "
bon
bon
"
```

Question 3 : En utilisant les fonctions `lignes`, `paragraphe` et `append`, écrire une définition de la fonction `colle-verticale2Par`, qui, étant donnés deux paragraphes `P1` et `P2`, rend le paragraphe formé à partir des lignes de `P1` et `P2`. Par exemple

```
(colle-verticale2Par
  (paragraphe '("bon" "jour"))
  (paragraphe '("ca" "va")))→
 "
bon
jour
ca
va
"
```

Question 4 : Écrire une fonction `colle-verticaleP`, qui, étant donnés un paragraphe `P` et un entier `n`, rend le paragraphe formé des lignes de `P` répétées `n` fois. Par exemple

```
 (colle-verticaleP (paragraphe '("bon" "jour")) 3)  → "
bon
jour
bon
jour
bon
jour
"
```

Question 5 : Écrire une fonction `indentationL`, qui, étant donnés une liste de lignes `lignes` et un entier `n`, rend la liste dont chaque ligne est indentée de `n` positions vers la droite. Par exemple

```
(indentationL '("bon" "jour") 5)  → ("     bon" "     jour")
```

Question 6 : Écrire une fonction `indentationP`, qui, étant donnés un paragraphe `P` et un entier `n`, rend le paragraphe indenté de `n` positions vers la droite. Par exemple

```
(indentationP (paragraphe '("bon" "jour")) 5)  → "
    bon
    jour
"
```

Question 7 : En utilisant les fonctions `indentationP` et `colle-verticale2Par`, écrire la définition de la fonction `par-decale-diag1`, qui étant donnés un paragraphe `P` et 2 entiers `k` et `d`; renvoie le paragraphe formé des lignes de `P` répétées `k` fois; et à chaque répétition les lignes sont décalées de `d` caractères selon la première diagonale. Par exemple

```
(par-decale-diag1 (paragraphe '("bonne" "journee")) 3 5)  → "
          bonne
          journee
     bonne
     journee
bonne
journee
"
```

Question 8 : Même question que précédemment en décalant selon la deuxième diagonale. La fonction sera nommée `par-decale-diag2`.

```
(par-decale-diag2 (paragraphe '("bonne" "journee")) 3 5)  → "
bonne
journee
     bonne
     journee
          bonne
          journee
"
```

Exercice 40 – Types Ligne et Paragraphe

Le but de cet exercice est d'implanter les fonctions sur les types `Ligne` et `Paragraphe`. On rappelle qu'une `Ligne` est une chaînes de caractères ne comportant pas de caractère de fin de ligne, et qu'un `Paragraphe` est une chaîne de caractères débutant par un caractère de fin de ligne, et suivi d'une suite de lignes, terminées chacune par un caractère de fin de ligne.

Pour manipuler les chaînes de caractères, vous utiliserez les fonctions `string-append`, `string-length` et `substring`, dont les spécifications sont données dans la carte de référence, ainsi que l'application `(string #\newline)` qui a pour valeur la chaîne de caractères constituée du seul caractère de fin de ligne.

Question 1 : Écrire la définition d'une fonction, nommée `paragrapheMoi`, qui transforme une liste de lignes en un paragraphe, selon la même spécification que celle donnée pour la fonction `paragraphe` dans la carte de référence.

```
(paragrapheMoi (list "bon" "jour"))  → "
bon
jour
"
```

Question 2 : Écrire la définition d'une fonction, nommée `paragraphe-consMoi`, qui rajoute une ligne en tête d'un paragraphe, selon la même spécification que celle donnée pour la fonction `paragraphe-cons` dans la carte de référence.

```
 (paragraphe-consMoi "bien le" (paragraphe '("bon" "jour")))  → "
bien le
bon
jour
"
```

Question 3 : Écrire la définition d'une fonction, nommée `lignesMoi`, qui transforme un paragraphe en une suite de ligne, selon la même spécification que celle donnée pour la fonction `lignes` dans la carte de référence.

Cette question est plus difficile que les précédentes, car il faut accéder aux *caractères* du paragraphe pour arriver à repérer chaque ligne (suite de caractères entre deux caractères de fin de ligne). De plus vous ne disposez pas de fonctions manipulant directement les caractères, et il faut donc traiter un caractères comme une *chaîne réduites à un seul caractère*. Par exemple l'application `(string #\newline)` a pour valeur la chaîne de caractères constituée du seul caractère de fin de ligne ; et l'application `(substring P i (+ i 1))` a pour valeur la chaîne de caractères constituée du caractère en position `i` dans la chaîne `P`.

```
 (lignesMoi (paragrapheMoi '("bien le" "bon" "jour")))  → ("bien le" "bon" "jour")
```

Exercice 41 – Dessin en paragraphe

Cet exercice fait travailler la structure de lignes et de paragraphes afin d'y bâtir une sorte de petite bibliothèque graphique.

Question 1 :

Écrire une fonction, nommée `diagonale2`, qui prend un nombre *n* et calcule un paragraphe formé de blancs et d'une diagonale descendante dessinée avec des étoiles. Ainsi

```
 (diagonale2 4) →

*
 *
  *
   *
```

Nota : on ne demande pas que toutes les lignes d'un paragraphe possèdent le même nombre de caractères.

Question 2 :

Écrire une fonction, nommée `diagonale1`, qui prend un nombre *n* et calcule un paragraphe formé de blancs et d'une diagonale ascendante dessinée avec des étoiles. Ainsi

```
 (diagonale1 4) →

   *
  *
 *
*
```

Nota : on ne demande pas que toutes les lignes d'un paragraphe possède le même nombre de caractères.

Question 3 :

Écrire une fonction, nommée `paragraphe-overlay`, prenant deux paragraphes provenant des fonctions précédentes et calculant un nouveau paragraphe superposition des deux arguments. La superposition

de deux blancs conduit à un blanc, toute autre superposition contenant au moins une étoile donne une étoile. Ainsi,

```
(paragraphe-overlay (diagonale2 4) (diagonale2 5)) →

*
 *
  *
   *
    *

(paragraphe-overlay (diagonale1 3) (diagonale2 5)) →

* *
 *
* *
   *
    *
```

Question 4 :

Écrire une fonction, nommée `paragraphe-normalise`, qui, à partir d'un paragraphe, calcule son normalisé. Un paragraphe normalisé est un paragraphe dans lequel aucune ligne ne comporte de blancs superflus en fin de ligne. Ainsi,

```
(lignes (paragraphe-normalise (paragraphe (list "  " "  * "))))  →
("" "  *")
```

6 Barrière d'abstraction

On introduit ici la notion de barrière d'abstraction.

Exercice 42 – Fiche de notes

Une *fi che de note* est une structure contenant les informations suivantes :

1. un numéro d'identification (type `nat`) ;
2. un nom (type `string`) ;
3. un prénom (type `string`) ;
4. une liste de notes (type `LISTE[Note]`, avec `Note = Nombre/entre 0 et 20/`).

On appelle `fiche-etudiant` le type de données ainsi défini.

On a défini une barrière d'abstraction permettant la création et l'accés aux informations contenues dans une fiche de notes.

```
;;; Barriere d'abstraction du type fiche-etudiant
;;; ============================================

;;; Constructeur
;;; ————————
;;; nouvelle-fiche : nat * string * string * LISTE[Note] -> fiche-etudiant
;;; avec Note = Nombre/entre 0 et 20/
;;; (nouvelle-fiche num nom prenom notes) renvoie une nouvelle fiche
;;; construite à partir des données passées en argument.
;;; ERREUR lorsque l'un des arguments n'est pas du bon type.

;;; Accesseurs
;;; ————-
```

```
;;; extract-ident : fiche-etudiant -> nat
;;; (extract-ident f) renvoie le numéro d'identification
;;; contenu dans la fiche f
;;; ERREUR lorsque f n'est pas de type fiche-etudiant.

;;; extract-nom : fiche-etudiant -> string
;;; (extract-nom f) renvoie le nom de l'etudiant
;;; contenu dans la fiche f
;;; ERREUR lorsque f n'est pas de type fiche-etudiant.

;;; extract-prenom : fiche-etudiant -> string
;;; (extract-prenom f) renvoie le prenom de l'etudiant
;;; contenu dans la fiche f
;;; ERREUR lorsque f n'est pas de type fiche-etudiant.

;;; extract-notes : fiche-etudiant -> LISTE[Note]
;;; avec Note = nat/entre 0 et 20/
;;; (extract-notes f) renvoie la liste de notes
;;; contenue dans la fiche f
;;; ERREUR lorsque f n'est pas de type fiche-etudiant.

;;; Reconnaisseur
;;; ————-
;;; fiche-etudiant? : Valeur -> bool
;;; (fiche-etudiant? x) renvoie #t si x est de type fiche-etudiant
;;; - c' est-a-dire: a ete cree avec le constructeur nouvelle-fiche -
;;; ou renvoie #f sinon.
```

Les valeurs de type `fiche-etudiant` sont abstraites, elle sont affichées sous la forme : `#<abstract:fiche-etudiant>`.

Question 1 :

Définir une fonction sans paramètre, nommée `fiche-abelin`, qui renvoie une fiche pour l'etudiant Jean ABELIN, dont le numéro d'identification est 3670, avec une liste de notes initialement vide.

Question 2 :

Écrire une fonction `ajoute-notes` qui prend en arguments une fiche et une liste de notes et qui rend une *nouvelle fi che* obtenue en ajoutant la liste de nouvelles notes à la fiche passée en premier argument. La fonction signalera une erreur si la liste de notes passée en argument n'est du type `LISTE[Note]`.

Exemples

```
(extract-notes (fiche-abelin))  → ()
```

On crée une nouvelle fiche avec une première liste de notes pour Abelin :

```
;;; fiche-abelin-bis :  -> fiche-etudiant
;;; (fiche-abelin-bis) renvoie une nouvelle fiche contenant plus de notes
;;; pour l'etudiant Jean ABELIN
(define (fiche-abelin-bis)
  (ajoute-notes (fiche-abelin) (list 11.5 15 9.5)))
```

On obtient :

```
(extract-notes (fiche-abelin-bis))  → (11.5 15 9.5)
```

On crée une troisième fiche qui augmente la liste de notes de la fiche précédente :

```
;;; fiche-abelin-ter : -> fiche-etudiant
;;; (fiche-abelin-ter) renvoie encore une nouvelle fiche contenant encore plus
;;;  de notes pour l'etudiant Jean ABELIN
(define (fiche-abelin-ter)
  (ajoute-notes (fiche-abelin-bis) (list 13 10.5 4)))
```

On obtient alors :

```
(extract-notes (fiche-abelin-ter))  → (13 10.5 4 11.5 15 9.5)
```

Question 3 :

Écrire une fonction `moyenne-etudiant` qui calcule la moyenne des notes contenues dans une fiche.

Remarque : l'exercice « Moyenne d'une liste de nombres » (page 13) a pour objet l'écriture d'une fonction qui calcule la moyenne des nombres contenus dans une liste.

Question 4 :

Les notes inférieures strictement à 5 sont éliminatoires. Écrire un prédicat qui vérifie que la liste de notes d'une fiche ne contient pas de note éliminatoire.

7 Problèmes

Les problèmes de cette section mettent en oeuvre toutes les notions présentées dans la saison 1.

Les problèmes « Base de données étudiants » (page 31) et « Base et tri » (page 31) utilisent la barrière d'abstraction introduite dans l'exercice « Fiche d'un étudiant » (page 25)

Exercice 43 – 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 (voir aussi l'exercice « Arbre de recherche d'un point dans un intervalle » (page **??**), où l'ensemble des points est représenté par un arbre).

Question 1 : 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 = NUPLET[Nombre Nombre]`.

Écrire le prédicat `estDansInterv?`, qui vérifie si un point *x* appartient à un intervalle *interv*. Par exemple

```
(estDansInterv? 5 '(8 12))  → #F
```

Question 2 : On suppose que l'ensemble des points est représenté par une liste (l'ordre des points dans la liste est quelconque). Écrire le semi-prédicat `rechercheListe` répondant à la spécification suivante :

```
;;; rechercheListe :  LISTE[Nombre] * Intervalle -> Nombre + #f
;;; avec Intervalle = NUPLET[Nombre Nombre]
;;; (rechercheListe L interv) renvoie un point de L (le plus près du début de la liste)
;; qui est dans l'intervalle interv, et renvoie #f si L ne contient pas de point dans interv.
;;; N.B. On ne suppose pas la liste L triée.
```

Par exemple

```
(rechercheListe '(2 4 12 6 10 5 9) '(5 8))  → 6
```

Question 3 : Pour aller plus loin

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

```
(define (rechercheListeFAUX L interv)  ; PROGRAMME INCORRECT
   (define (rechL-x1-x2 Liste)
     (if (pair? Liste)
         (let ((valeur (car Liste)))
           (if (and (>= valeur x1) (<= valeur x2))
               valeur
               (rechL-x1-x2 (cdr Liste))))
         #f))
   (let ((x1 (car interv))
         (x2 (cadr interv)))
     (rechL-x1-x2 L) ))
```

Par exemple
```
(rechercheListeFAUX '( 2 12 6 5 9) '(5 8))
  ->  reference to undefined identifier: x1
```
Rechercher pourquoi la fonction `rechercheListeFAUX` n'est pas correcte, et la corriger en une fonction `rechercheListeJUSTE`.

Question 4 : On suppose à présent que les points de la liste sont triés en ordre croissant de leurs abscisses. Écrire le semi-prédicat `rechercheListeTriee` répondant à la spécification suivante :

`;;;` *rechercheListeTriee : Intervalle * LISTE[Nombre] -> Nombre + #f*
`;;;` *HYPOTHESE : la liste L est triée*
`;;;` *(rechercheListeTriee Int L) renvoie ... (voir question 2)*

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 8).
```
(rechercheListeTriee '(2 3 12 20 7 25) '(5 8))  → #F
(rechercheListeTriee '(1 2 9 6 3 10 15) '(8 12))  → 9
(rechercheListeTriee '(1 2 10 9 15 20) '(8 12))  → 10
```

Exercice 44 – Y-a-t-il un point dans le rectangle ?

On traite ici le même problème que dans l'exercice « Y-a-t-il un point dans l'intervalle ? » (page 27), 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 = NUPLET[Nombre Nombre]`, et
– `Rectangle = NUPLET[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 fonction `(estDansRect? P Rect)` qui renvoie `Vrai` si et seulement si le point `P` appartient, au sens large, au rectangle `Rect`. Par exemple
```
(estDansRect? '(3 5) '((2 3) (5 10)))  → #T
```

Question 3 : L'ensemble des points est représenté par une liste.
Écrire le 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
;;; près 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))
  '((6 8) (-1 -1) (2 6) (3 4) (2 3)))→
 (3 4)
```

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 :

```
;;; HYPOTHESE : La liste de points est triée selon les abscisses
```

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 « Y-a-t-il un point dans l'intervalle ? » (page 27).

Exercice 45 – Base de données d'étudiants

Cet exercice fait suite à l'exercice « Fiche de notes » (page 25).

Dans cet exercice, une *base de données d'étudiants* est une liste de fiches de type `fiche-etudiant`, rangées par ordre croissant de numéro d'identification.

Question 1 : Définir des fonctions sans paramètres, nommées `fiche-abelin, fiche-lesueurC, fiche-dupond, fiche-zidi, fiche-lesueurM, fiche-fong` qui renvoient les fiches contenant les renseignements suivants :

numero	nom	prenom	notes
3670	ABELIN	Jean	4 12 10
3987	LESUEUR	Claude	13 7.5 10.5
4530	DUPOND	Paul	10 12 10.5 11.5
3698	ZIDI	Marie	9 12 10
4980	LESUEUR	Marie	12.5 12 8
3940	FONG	Jean	14 16

Consigne : écrire le nom en majuscules, écrire l'initiale du prénom en majuscule et le reste du prénom en minuscules, écrire la liste des notes dans l'ordre où elles sont données.

Question 2 : Écrire une fonction `affichage-etudiant` qui, étant donnée une fiche retourne la ligne formée du numéro d'identification, du nom, du prénom et de la liste de notes de l'étudiant. Par exemple :

```
(affichage-etudiant (fiche-abelin)) -> "3670 ABELIN Jean (4 12 10)"
```

Consigne : les différentes rubriques (numéro d'identification, nom, prénom, liste de notes) doivent être séparées par un espace.

Question 3 : Écrire une fonction `affichage-liste-etudiants` qui, étant donnée une liste `L` de fiches de type `fiche-etudiant`, retourne le paragraphe dont chaque ligne correspond à l'affichage d'un étudiant. Par exemple :

```
(affichage-liste-etudiants (list (fiche-abelin)
                                 (fiche-lesueurM)
                                 (fiche-fong))) →
"
3670 ABELIN Jean (4 12 10)
4980 LESUEUR Marie (12.5 12 8)
3940 FONG Jean (14 16)
"
```

Question 4 : Écrire une fonction `insertion-base` qui, étant données une fiche et une base de données d'étudiants, retourne la base de données obtenue en insérant la fiche à sa place dans la base de données initiale. On suppose que le numéro d'identification de la fiche à insérer n'est pas déjà présent dans la base de données.

Question 5 : Utilisez la fonction `insertion-base` pour définir une fonction sans paramètres, nommée `base-exemple` qui renvoie la base de données d'étudiants constituée des fiches `fiche-abelin`, `fiche-lesueurC`, `fiche-dupond`, `fiche-zidi`, `fiche-lesueurM`, `fiche-fong`.
Si vous appliquez la fonction `affichage-liste-etudiants` à la fonction `base-exemple`, vous devez obtenir le résultat suivant :

```
"
3670 ABELIN Jean (4 12 10)
3698 ZIDI Marie (9 12 10)
3940 FONG Jean (14 16)
3987 LESUEUR Claude (13 7.5 10.5)
4530 DUPOND Paul (10 12 10.5 11.5)
4980 LESUEUR Marie (12.5 12 8)
"
```

Question 6 : Écrire une fonction `recherche-base` qui, étant donnés un entier *n* et une base de données d'étudiants, retourne la fiche de numéro d'identification *n*, s'il existe une telle fiche dans la base de données, et retourne `#f` sinon.

Question 7 : Écrire une fonction `arrondi-un` qui, étant donné un nombre, retourne sa valeur arrondie à une décimale. Par exemple :

```
(arrondi-un 12.82)  → 12.8
(arrondi-un 12.87)  → 12.9
```

Vous pourrez utiliser les fonctions `round` et `exact->inexact` (cherchez leurs spécifications dans Scheme, par exemple dans le menu Help de DrScheme).

Question 8 : Écrire une fonction `affichage-resultats-etudiant` qui, étant donnée une fiche retourne la ligne formée du nom, du prénom de l'étudiant, de sa moyenne arrondie à une décimale et de sa mention. Par exemple :

```
(affichage-resultats-etudiant (fiche-zidi)) →
 "ZIDI Marie 10.3 mention P"
```

Consigne : les différentes rubriques (numéro d'identification, nom, prénom, liste de notes) doivent être séparées par un espace.

Vous pouvez utiliser la fonction `moyenne-etudiant` vue dans l'exercice « Fiche de notes » (page 25) et la fonction `mention` vue dans l'exercice « Calcul des mentions » (page **??**).

Question 9 : Écrire une fonction `attestation-de-notes` qui, étant donnés un numéro d'identification et une base de données d'étudiants, retourne la ligne formée du nom, du prénom, de la moyenne

arrondie à une décimale et de la mention de l'étudiant correspondant au numéro d'identification. La fonction doit signaler une erreur si une telle fiche n'existe pas. Par exemple :

```
(attestation-de-notes 3987 (base-exemple)) →
"LESUEUR Claude 10.3 mention P"

(attestation-de-notes 4000 (base-exemple)) →
```
attestation-de-notes : ERREUR : le numero 4000 n'existe pas

Question 10 : Écrire une fonction `moyenne-promotion` qui, étant donnée une base de données d'étudiants, calcule la moyenne générale de la promotion, arrondie à une décimale. Par exemple :

```
(moyenne-promotion (base-exemple)) -> 11.0
```

Question 11 : Écrire une fonction `affichage-des-recus` qui, étant donnée une base de données d'étudiants, retourne le paragraphe dont les lignes correspondent à l'affichage des noms et prénoms des étudiants ayant au moins 10 de moyenne.

Question 12 : Écrire une fonction `nombre-recus` qui, étant donnée une base de données d'étudiants, retourne le nombre d'étudiants ayant au moins 10 de moyenne.

Exercice 46 – Base et tri

Cet exercice fait suite à l'exercice « Fiche de notes » (page 25) et à l'exercice « Base de données d'étudiants » (page 31).

Comme dans l'exercice « Base de données d'étudiants » (page 31) une *base de données d'étudiants* est une liste de fiches de type `fiche-etudiant`, rangées par ordre croissant de numéro d'identification. On rappelle que, dans cet exercice, on a défini des fonctions sans paramètres, nommées `fiche-abelin`, `fiche-lesueurC`, `fiche-dupond`, `fiche-zidi`, `fiche-lesueurM`, `fiche-fong` qui renvoient les fiches contenant les renseignements suivants :

numero	nom	prenom	notes
3670	ABELIN	Jean	4 12 10
3987	LESUEUR	Claude	13 7.5 10.5
4530	DUPOND	Paul	10 12 10.5 11.5
3698	ZIDI	Marie	9 12 10
4980	LESUEUR	Marie	12.5 12 8
3940	FONG	Jean	14 16

On désire faire des affichages de bases de données par ordre alphabétique ou par ordre de mérite.

Question 1 : On considère un ensemble E muni d'une relation d'ordre (ou de pré-ordre). S'inspirer de l'exercice « Tri fusion » (page 18) pour définir une fonction `tri-fusion-generique` qui prend en arguments un prédicat et une liste d'éléments de E et qui retourne la liste rangée dans l'ordre défini sur E. La fonction n'éliminera pas les doublons. Par exemple :

```
(tri-fusion-generique > '(1 6 2 19 3 13 11)) → (19 13 11 6 3 2 1)
(tri-fusion-generique < '(1 6 2 19 3 13 11)) → (1 2 3 6 11 13 19)
(tri-fusion-generique < '(1 3 6 19 3 13 11)) → (1 3 3 6 11 13 19)
(tri-fusion-generique string<?
                  '("chien" "chat" "lapin" "cheval" "chameau"))
                → ("chameau" "chat" "cheval" "chien" "lapin")
```

Question 2 : Écrire une fonction `liste->base` qui, étant donnée une liste de fiches de type `fiche-etudiant`, retourne la base de données obtenue en rangeant les fiches par ordre croissant de numéro d'identification.

Question 3 : Utilisez la fonction `liste->base` pour définir une fonction sans paramètres, nommée `base-exemple` qui renvoie la base de données d'étudiants constituée des fiches `fiche-abelin`, `fiche-lesueurC`, `fiche-dupond`, `fiche-zidi`, `fiche-lesueurM`, `fiche-fong`.

Question 4 : Écrire une fonction `affichage-resultats-alphabetique` qui, étant donnée une base de données d'étudiants, retourne le paragraphe dont chaque ligne est formée du nom, du prénom de la moyenne arrondie à une décimale et de la mention de chaque étudiant de la base de données. L'affichage doit se faire dans l'ordre alphabétique. Par exemple :

```
(affichage-resultats-alphabetique (base-exemple)) →
"
ABELIN Jean 8.7 ajourné
DUPOND Paul 11.0 mention P
FONG Jean 15.0 mention B
LESUEUR Claude 10.3 mention P
LESUEUR Marie 10.8 mention P
ZIDI Marie 10.3 mention P
"
```

Vous pourrez utiliser la fonction `string<?` (cherchez sa spécification dans Scheme) et la fonction `affichage-resultats-etudiant` « Base de données d'étudiants » (page 31)

Question 5 : Écrire une fonction `affichage-resultats-merite` qui, étant donnée une base de données d'étudiants, retourne le paragraphe dont chaque ligne est formée du nom, du prénom de la moyenne arrondie à une décimale et de la mention de chaque étudiant de la base de données. L'affichage doit se faire dans l'ordre décroissant des moyennes ; les étudiants ayant la même moyenne seront rangés dans l'ordre alphabétique. Par exemple :

```
(affichage-resultats-merite (base-exemple)) →
"
FONG Jean 15.0 mention B
DUPOND Paul 11.0 mention P
LESUEUR Marie 10.8 mention P
LESUEUR Claude 10.3 mention P
ZIDI Marie 10.3 mention P
ABELIN Jean 8.7 ajourné
"
```

Question 6 : Écrire une fonction `affichage-des-recus-alphabetique` qui, étant donnée une base de données d'étudiants, retourne le paragraphe dont les lignes correspondent à l'affichage des noms et prénoms des étudiants ayant au moins 10 de moyenne. Par exemple :

```
(affichage-des-recus-alphabetique (base-exemple)) →
"
DUPOND Paul
FONG Jean
LESUEUR Claude
LESUEUR Marie
ZIDI Marie
"
```

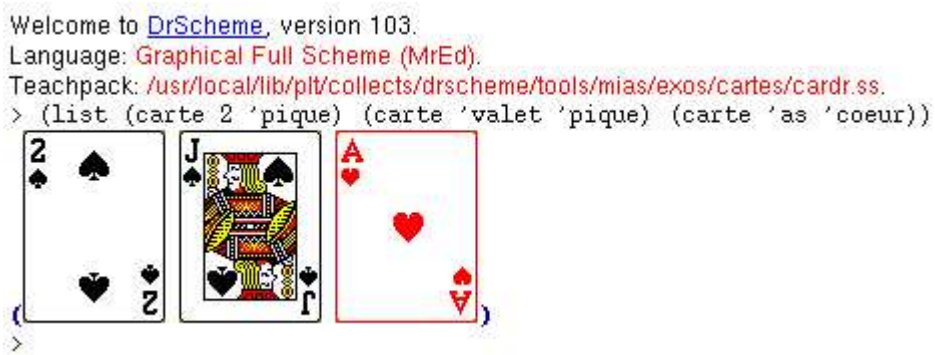
Exercice 47 – Cartes à jouer

Cet exercice manipule des cartes à jouer. Un paqueçon (c'est ce que DrScheme nomme un *teachpack*) particulier procure les fonctions suivantes dont les spécifications suivent :

```
;;; carte: Rang * Couleur -> Carte
;;; (carte rang couleur) rend la carte ainsi nommée. Rang et couleur
;;; sont définis comme les valeurs de retour des accesseurs
;;; carte-rang et carte-couleur.

;;; carte-couleur: Carte -> Couleur
;;; (carte-couleur c) rend la couleur (un Symbole) de la carte à savoir: pique,
;;; coeur, carreau ou trefle.
```

;;; carte-rang: Carte -> Rang
;;; (carte-rang c) rend le Rang de la carte (un nombre ou un Symbole) à savoir:
;;; as, 2, 3, 4, 5, 6, 7, 8, 9, 10, valet, dame ou roi.
Voici un aperçu des cartes de DrScheme :



Question 1 : Le bridge est un jeu de cartes qui compare les couleurs comme suit : pique est plus fort que coeur qui est plus fort que carreau qui est plus fort que trèfle.

Écrire une fonction nommée `couleur>` prenant deux couleurs et testant si la première est strictement plus forte que la seconde.

Question 2 : Écrire la fonction nommée `couleur>=` qui prend deux couleurs et teste si la première est plus forte que ou égale à la seconde.

Question 3 : Au bridge, à l'intérieur d'une couleur, les rangs des cartes sont comparés ainsi : l'as est plus fort que le roi qui est plus fort que la dame qui est plus forte que le valet qui est plus forte que le 10 qui est plus fort que le 9 . . .qui est plus fort que le 3 qui est plus fort que le 2.

Écrire les fonctions nommées `rang>` et `rang>=` comparant les rangs des cartes.

Question 4 : Écrire maintenant une fonction nommée `bridge>=` qui prend deux cartes et vérifie que la première est plus forte que ou égale à la seconde.

Question 5 : Écrire une fonction nommée `paquet` qui construit la liste (le paquet) de toutes les cartes. Un jeu de 52 cartes bien entendu ! Les cartes seront ordonnées en commençant par la plus forte de toutes.

Question 6 : Écrire la fonction nommée `coupe` qui prend une liste de cartes que l'on nommera dorénavant un paquet et un entier naturel *n* entre 0 et le nombre de cartes présentes dans le paquet. Le résultat est une nouveau paquet de cartes dont les *n* dernières sont les *n* qui se trouvaient en tête du paquet initial.

Ainsi si l'on a un paquet de cartes (a, b, c, d, e, f, g), le couper après 3 donnera le paquet (d, e, f, g, a, b, c).

Question 7 : Écrire la fonction nommée `melange-par-coupes` qui prend un entier naturel *n* et un paquet et le coupe *n* fois (afin que les cartes soient en ordre dispersé). On s'aidera de la fonction suivante pour obtenir des nombres aléatoires (ou considérés comme tels) :

;;; random: nat -> nat
;;; (random n) rend un nombre aléatoire compris entre 0 (inclus) et n (exclus).
Expérience : couper 52 fois un paquet de 52 cartes puis examiner le résultat.

Question 8 : Le mélange précédent n'est pas terrible. Écrire une nouvelle fonction nommée `melange-par-entrelacement` qui sépare le paquet en deux et le réassemble en faisant que chaque carte d'un demi-paquet est glissée entre chaque carte du second demi-paquet.

Ainsi si l'on a un paquet de cartes (a, b, c, d, e, f, g) cette opération conduira aux deux paquets (a, b, c, d) et (e, f, g) puis à les entrelacer ce qui donnera (a, e, b, f, c, g, d).

Question 9 : On peut également mélanger en distribuant les cartes en *n* paquets que l'on entasse après en un unique paquet. Écrire une fonction nommée `melange-par-distribution` qui prend un naturel et un paquet et qui réalise ce mélange.

Ainsi si l'on a un paquet de cartes (a, b, c, d, e, f, g) et que l'on distribue en trois tas, on obtient un premier tas contenant (a, d, g), un second tas contenant (b, e) et un troisième tas contenant (c, f). Le résultat final sera donc (a, d, g, b, e, c, f) puisque l'on entasse les tas dans l'ordre où on les a créés.

Question 10 : Combiner ces trois mélanges, écrire donc la fonction `melange` qui prend un paquet de cartes et le mélange en effectuant des mélanges des trois types.

Question 11 : Maintenant que les paquets sont bien mélangés, écrire une fonction, nommée `tri`, prenant un paquet et un prédicat de comparaison (comme la fonction `bridge>=`) et rendant un paquet ordonné suivant ce prédicat c'est-à-dire un paquet commençant d'abord par la plus forte carte.

On utilisera pour ce faire la méthode dite du tri fusion. La fusion consiste à prendre deux paquets de cartes triées correctement et à les fusionner en un unique paquet. Il suffit pour ce faire de ne considérer que la première carte de chaque paquet, de prendre la plus forte et de recommencer jusqu'à l'épuisement des deux paquets.

Le tri d'un paquet formé d'une seule carte est trivial : il n'y a rien à faire ! Lorsqu'un paquet est plus gros, on le coupe en deux sous-paquets, on trie chacun des sous-paquets et l'on fusionne les deux paquets résultant.

Exercice 48 – Mon beau sapin

Le but de cet exercice est de construire des sapins à partir de caractères simples dans le genre de ceux qui suivent :

```
(sapin 6) →

    /\
   /%%\
  /%%%%\
 /%%%%%%\
/%%%%%%%%\
    ||
```

Le fichier qui suit contient les définitions que vous aurez à lire et utiliser. En particulier, il contient les dessins de base et les fonctions permettant de coller horizontalement ou verticalement les images produites. Dans tout ce qui suit, vous ne devez jamais utiliser de chaînes de caractères directement, vous êtes contraints d'employer cette couche d'abstration manipulant des images et les collant afin d'en constituer de plus grandes.

```
;;; Id : sapin.scm,v 1.12001/06/3017 : 33 : 33 queinnecExp
;;; Copyright (C) 2001 by Christian.Queinnec@lip6.fr

;;; Ce fichier contient les définitions de base à utiliser pour l'exercice
;;; concernant les sapins textuels.

;;; tronc: -> Image
;;; (tronc) renvoie l'image d'un tronc de sapin.

(define (tronc)
  "|" )
```

```
;;; gauche: -> Image
;;; (gauche) renvoie l'image d'une frondaison gauche.

(define (gauche)
  "/" )

;;; droite: -> Image
;;; (droite) renvoie l'image d'une frondaison droite.

(define (droite)
  "\\" )

;;; milieu: -> Image
;;; (milieu) renvoie l'image d'un petit morceau de ramure de sapin décoré
;;; de boules de noël.

(define (milieu)
  "%" )

;;; blanc: -> Image
;;; (blanc) renvoie une image rectangulaire blanche.

(define (blanc)
  " " )

;;; vide: -> Image
;;; (vide) renvoie une image vide de largeur nulle.

(define (vide)
  "" )

;;; NOTE: Attention ces collages sont rudimentaires et mal abstraits.
;;; C'est normal, ils font l'objet de questions.

;;; collage-vertical: Image * Image -> Image
;;; (collage-vertical image1 image2) renvoie une nouvelle image composée,
;;; pour sa partie supérieure d'image1 et, pour sa partie inférieure, d'image2.

(define (collage-vertical image1 image2)
  (define (a-la-ligne)
    (string #\newline) )
  (string-append image1 (a-la-ligne) image2) )

;;; collage-horizontal: Image * Image -> Image
;;; (collage-horizontal image1 image2) renvoie une nouvelle image composée,
;;; pour sa partie gauche d'image1 et, pour sa partie droite, d'image2.

(define (collage-horizontal image1 image2)
  (string-append image1 image2) )

;;; end of sapin.scm
```

Question 1 : La fonction `collage-horizontal` a été écrite à la va-vite. Quelle restriction faut-il ajouter à la spécification pour qu'elle fasse ce qui est souhaité ?

Écrire une fonction nommée `mauvais-collage` collant horizontalement deux dessins dont le résultat n'est pas celui que l'on pourrait attendre.

Question 2 : À l'aide des fonctions précédentes, écrire des fonctions menant aux dessins suivants.

Chaque dessin est préfixé de l'appel qui le construit.

```
   (barriere) →

| | |

   (petit-losange) →

/\
\/

   (grand-losange) →

 /\
//\\
\\//
 \/
```

Question 3 : Écrire une fonction nommée `repetition` prenant une image *i* et un entier naturel *n* et retournant une nouvelle image composée horizontalement de *n* fois l'image *i*.

Question 4 : Écrire une fonction nommée `bordage` prenant trois images respectivement nommées *g*, *c* et *d* ainsi qu'un nombre naturel *i*. La fonction construit l'image $gg\ldots gcdd\ldots d$ où *g* et *d* sont répétés *i* fois.

Question 5 : On différencie dans un sapin la ramure (qui s'étend sur plusieurs lignes) du tronc qui lui tient sur la dernière ligne. Écrire la fonction nommée `etage` qui construit un étage de ramure. Cette fonction prendra deux entiers naturels. Le premier décrit l'épaisseur d'un demi-sapin, le second la moitié de la largeur de l'image à produire. Pour comprendre ces nombres, voici des exemples d'étage :

```
   (etage 1 5) →

    /\

   (etage 1 6) →

     /\

   (etage 3 5) →

  /%%%%\

   (etage 3 6) →

   /%%%%\

   (etage 3 7) →

    /%%%%\

   (etage 5 5) →

/%%%%%%%%\
```

Question 6 : Écrire une fonction nommée `sapin` qui prend un entier naturel strictement positif et construit un sapin de cette hauteur (tronc compris). On pourra s'aider d'une fonction annexe construisant la ramure.

Table des matières