Exercices de programmation en CAML

niveau classes préparatoires

http://alexandre-mesle.com

6 septembre 2011

Table des matières

Chapitre 1

Techniques de programmation

1.1 Initiation

1.1.1 Factorielle

On définit la fonction factorielle par récurrence de la façon suivante :

$$\begin{cases} n! = n.((n-1)!) \\ 0! = 1 \end{cases}$$

Exercice 1

Écrire la fonction `factorielle : int -> int` qui à n associe $n!$.

Exercice 2

Écrire la fonction `factorielle_acc : int -> int -> int` telle que `factorielle_acc n k` calcule $k(n!)$

Exercice 3

Écrire une fonction `factorielle_bis : int -> int` qui invoque `factorielle_acc` de sorte que `factorielle_bis n` calcule $n!$.

1.1.2 Exponentiation lente

L'élévation de b à la puissance n se définit récursivement par itération du produit

$$\begin{cases} b^n = b.b^{n-1} \\ b^0 = 1 \end{cases}$$

Exercice 4

Écrire une fonction `slow_exp : int -> int -> int` prenant b et n en paramètres en calculant b^n à l'aide de la relation de récurrence mentionnée ci-dessus.

Exercice 5

Écrire une fonction recursive terminale `slow_exp_acc : int -> int -> int -> int` prenant a, b et n en paramètres en calculant $a.b^n$.

Exercice 6

Écrire une fonction `slow_exp_bis : int -> int -> int` qui invoque `slow_exp_acc` de sorte que `slow_exp_bis b n` calcule b^n.

1.1.3 Exponentiation rapide

Nous allons calculer b^n à l'aide des relations de récurrence suivantes

$$\begin{cases} b^n = (b^{n/2})^2, \ \textit{si n est pair} \\ b^n = b.b^{n-1}, \ \textit{si n est impair} \\ b^0 = 1 \end{cases}$$

Exercice 7

Ecrire une fonction `even : int -> bool` telle que `even n` retourne `true` si et seulement si n est pair.

Exercice 8

Définir la fonction `fast_exp : int -> int -> int` telle que `fast_exp b n` calcule b^n avec les égalités ci-dessus.

Exercice 9

Prouvez que la fonction `fast_exp` se termine toujours.

Exercice 10

Donnez une version récursive terminale de `fast_exp`.

1.1.4 PGCD

On définit le plus grand commun diviseur (greatest common divider) de deux entiers récursivement de la façon suivante :

$$\begin{cases} gcd(n, m) = gcd(m, n \ mod \ m) \\ gcd(n, 0) = n \end{cases}$$

Exercice 11

Prouvez que cette relation de récurrence permet d'écrire une fonction qui se termine toujours.

Exercice 12

Définir la fonction `gcd : int -> int -> int` telle que `gcd n m` calcule le PGCD de n et de m à l'aide des relations ci-dessus.

1.1.5 Caractères et entiers

Exercice 13

Écrire la fonction `lettre_suivante : char -> char` telle que `lettre_suivante a` retourne la lettre située après a dans l'alphabet. Nous ne gérerons pas le cas particulier où `a = 'z'`.

Exercice 14

Écrire la fonction `figure_of_int : int -> char` telle que `figure_of_int n` retourne le caractère représentant le chiffre n. Par exemple, `figure_of_int 4` retourne `'4'`. Nous ne gérerons pas les cas particuliers où `n` est un nombre de plusieurs chiffres.

Exercice 15

Écrire la fonction `compare_lettres : char -> char -> bool` telle que `compare_lettres a b` retourne vrai ssi la lettre a et située avant la lettre b dans l'alphabet ou si les deux lettres sont les mêmes.

Exercice 16

Écrire la fonction `alphabet : char -> char -> unit` telle que `alphabet lettre_depart lettre_arrivee` affiche l'alphabet entre les lettres `lettre_depart` et `lettre_arrivee`.

1.1.6 Mise sous forme récursive

Exercice 17

Écrire la fonction `mult : int -> int -> int` telle que `mult x y` retourne le produit de x par y sans effectuer de multiplication.

Exercice 18

Mettez la fonction précédente sous forme récursive terminale. Encapsulez-là dans une fonction `mult_bis : int -> int -> int`.

1.1.7 Indicatrice d'Euler

Exercice 19

Écrire une fonction `ind_euler : int -> int` retournant l'indicatrice d'Euler de l'entier passé en paramètre.

1.2 Listes

Exercice 1

Écrire la fonction `n_a_un : int -> int list` retournant la liste $[n; \ldots; 2; 1]$.

Exercice 2

Écrire la fonction `un_a_n : int -> int list` retournant la liste $[1; 2; \ldots; n]$.

Exercice 3

Écrire la fonction `supprime : 'a list -> int -> 'a list` retournant dans laquelle les n premiers éléments ont été supprimés.

Exercice 4

Écrire la fonction `long_prefixe : 'a list -> int` retournant le nombre d'entiers identiques se trouvant au début de la liste passée en paramètre.

1.3 Types récursifs

1.3.1 Tri par arbre binaire de recherche

Un ABR (arbre binaire de recherche) est un arbre tel que pour chaque noeud x, tous les éléments du sous-arbre gauche ont une valeur inférieure à x et tous les éléments du sous-arbre droit ont une valeur supérieure à x. Nous utiliserons dans les exercices suivants le type

```
type 'a bin_tree =
    Empty
    | Node of 'a bin_tree * 'a * 'a bin_tree;;
```

Exercice 1

Écrire une fonction `ins_abr : ('a -> 'a -> bool) -> 'a -> 'a bin_tree -> 'a bin_tree` telle que `ins_abr f x t` insère l'élément x dans l'arbre t en utilisant la fonction de comparaison f.

Exercice 2

Écrire une fonction `abr_of_list : ('a -> 'a -> bool) -> 'a list -> 'a bin_tree` telle que `abr_of_list f l` retourne un arbre contenant tous les éléments de l, disposés avec la fonction f.

Exercice 3

Écrire la fonction `list_of_abr : 'a bin_tree -> 'a list` retournant une liste contenant les éléments de l'arbre passé en paramètre.

Exercice 4

Écrire une fonction `tri_liste : ('a -> 'a -> bool) -> 'a list -> 'a list` qui trie la liste passée en paramètre à l'aide d'un ABR.

1.3.2 Fonctions

Nous utiliserons le type suivant pour représenter des fonctions simples :

```
type exp =
    Const of int
  | X
  | Somme of exp * exp
  | Produit of exp * exp
  | Puiss of exp * int;;
```

Exercice 5

Écrire une fonction `str_of_exp : exp -> string` telle que `str_of_exp e` retourne une représentation sous forme de chaîne de caractères de l'expression e.

Exercice 6

Écrire une fonction `image : exp -> int -> int` telle que `image e x` retourne l'image de x par la fonction e.

Exercice 7

Écrire une fonction `derive : exp -> exp` retournant la dérivée de l'expression passée en paramètre.

Exercice 8

Écrire une fonction `degre : exp -> int` retournant le degré du polynôme représenté par l'expression passée en paramètre.

Chapitre 2

Problèmes d'algorithmique

2.1 Le carré qui rend fou

2.1.1 Affichage de carres

Nous souhaitons afficher sur la console des carrés de la façon suivante :

```
# print_string (carre 5);;
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
```

Exercice 1

Écrire la fonction `ligne : int -> string` retournant n points séparés par des espaces avec un retour à la ligne à la fin. Par exemple,

```
# ligne 6;;
- : string = ". . . . . . \n"
```

Exercice 2

Écrire la fonction `carre : int -> string` retournant un carré un coté n avec deux retours à la ligne à la fin. Par exemple,

```
# carre 6;;
- : string =
". . . . . . \n. . . . . . \n. . . . . . \n. . . . . . \n. . . . . . \n. . . . . . \n\n"
```

2.1.2 Remplissage avec des étoiles

Nous souhaitons maintenant afficher un carré contenant un rectangle d'étoiles. Par exemple,

```
# print_string (carre_etoiles (2, 8, 4, 9) 10);;
. . . . . . . . . .
. . . * * * * * * .
. . . * * * * * * .
. . . * * * * * * .
. . . * * * * * * .
. . . * * * * * * .
```

```
. . . * * * * * * .
. . . * * * * * * .
. . . . . . . . . .
. . . . . . . . . .
```

Exercice 3

Écrire la fonction `etoiles : int -> int -> int -> string` telle que `etoiles a b n` retourne n caractères séparés par des espaces avec un retour à la ligne à la fin, et dont les caractères dont l'indice se trouve entre a et b sont des étoiles. Par exemple,

```
# etoiles 3 8 10;;
- : string = ". . * * * * * * . . \n"
```

Exercice 4

Écrire la fonction `carre_etoiles : int * int * int * int -> int -> string` telle que `carre_etoiles (lhaut, lbas, cgauche, cdroite) n` retourne un carré un coté n contenant un rectangle d'étoiles dont le sommet en haut à gauche admet pour coordonnées $(cgauche, lhaut)$ et le sommet en bas à droite $(cdroite, lbas)$.

2.1.3 Enumération des carrés

Nous souhaitons afficher tous les carrés d'étoiles possible. Par exemple,

```
# print_str_list (tous_les_carres 3);;
* . .
. . .
. . .

. . .
* . .
. . .

. . .
. . .
* . .

* * .
* * .
. . .

. . .
* * .
* * .

* * *
* * *
* * *

. * .
. . .
. . .

. . .
. * .
```

```
. . .

. . .
. . .
. * .

. * *
. * *
. . .

. . .
. * *
. * *

. . *
. . .
. . .

. . .
. . *
. . .

. . .
. . .
. . *
```

Exercice 5

Écrire la fonction `couples_coordonnees_a_fixe : int -> int -> (int * int) list` telle que `couples_coordon` `a b` retourne tous les couples (a, y) vérifiant $a \leq y \leq b$. Par exemple,

```
# couples_coordonnees_a_fixe 3 8;;
- : (int * int) list = [(3, 3); (3, 4); (3, 5); (3, 6); (3, 7); (3, 8)]
```

Combien de couples sont retournés par cette fonction ?

Exercice 6

Écrire la fonction `couples_coordonnees : int -> (int * int) list` telle que `couples_coordonnees n` retourne tous les couples (x, y) vérifiant $1 \leq x \leq y \leq n$. Par exemple,

```
# couples_coordonnees 3;;
- : (int * int) list = [(1, 1); (1, 2); (1, 3); (2, 2); (2, 3); (3, 3)]
```

Combien de couples sont retournés par cette fonction ?

Exercice 7

Écrire la fonction `coordonnees_carres_colonnes_fixees : int * int -> int -> (int * int * int * int) list` telle que `coordonnees_carres_colonnes_fixees (cgauche, cdroite) n` retourne les coordonnées de tous les carrés d'étoiles dont les colonnes ont les indices $(cgauche, cdroite)$. Par exemple,

```
# coordonnees_carres_colonnes_fixees (3, 5) 7;;
- : (int * int * int * int) list =
[(1, 3, 3, 5); (2, 4, 3, 5); (3, 5, 3, 5); (4, 6, 3, 5); (5, 7, 3, 5)]
```

Combien d'éléments contient la liste retournée par cette fonction ?

Exercice 8

Écrire la fonction `coordonnees_carres : int -> (int * int * int * int) list` telle que `coordonnees_carres n` retourne les coordonnées de tous les carrés d'étoiles qu'il est possible de former dans un carré de côté n. Par exemple,

```
#   coordonnees_carres 3;;
- : (int * int * int * int) list =
[(1, 1, 1, 1); (2, 2, 1, 1); (3, 3, 1, 1); (1, 2, 1, 2); (2, 3, 1, 2);
 (1, 3, 1, 3); (1, 1, 2, 2); (2, 2, 2, 2); (3, 3, 2, 2); (1, 2, 2, 3);
 (2, 3, 2, 3); (1, 1, 3, 3); (2, 2, 3, 3); (3, 3, 3, 3)]
```

Combien d'éléments contient la liste retournée par cette fonction ?

Exercice 9

Définir une fonction `tous_les_carres : int -> string list` telle que `tous_les_carres n` retourne un liste contenant tous les carrés d'étoiles au format chaîne de caractère. Par exemple,

```
# tous_les_carres 3;;
- : string list =
["* . . \n. . . \n. . . \n\n"; ". . . \n* . . \n. . . \n\n";
 ". . . \n. . . \n* . . \n\n"; "* * . \n* * . \n. . . \n\n";
 ". . . \n* * . \n* * . \n\n"; "* * * \n* * * \n* * * \n\n";
 ". * . \n. . . \n. . . \n\n"; ". . . \n. * . \n. . . \n\n";
 ". . . \n. . . \n. * . \n\n"; ". * * \n. * * \n. . . \n\n";
 ". . . \n. * * \n. * * \n\n"; ". . * \n. . . \n. . . \n\n";
 ". . . \n. . * \n. . . \n\n"; ". . . \n. . . \n. . * \n\n"]
```

Exercice 10

Écrire la fonction `print_str_list : string list -> unit` telle que `print_str_list l` affiche toutes les chaînes de la liste l les unes à la suite des autres. Par exemple,

```
# print_str_list ["debut " ; "milieu et " ; "fin"];;
debut milieu et fin
- : unit = ()
```

2.1.4 Enumération des rectangles

Nous souhaitons afficher tous les rectangles d'étoiles possible. Par exemple,

```
# print_str_list (tous_les_rectangles 2);;
* .
. .

* *
. .

. *
. .

* .
* .

* *
* *
```

```
. *
. *

. .
* .

. .
* *

. .
. *
```

Exercice 11

Définir une fonction `concat_produits : 'a * 'b -> ('c * 'd) list -> ('a * 'b * 'c * 'd)` telle que `concat_produits (a, b) l` retourne une liste de quadruplets dont les deux premiers éléments sont a et b et les deux suivants pris dans la liste l. Par exemple,

```
# concat_produits (2, 4) [(1,2);(3,4);(5,6);(7,8);(9,0)];;
- : (int * int * int * int) list =
[(2, 4, 1, 2); (2, 4, 3, 4); (2, 4, 5, 6); (2, 4, 7, 8); (2, 4, 9, 0)]
```

Exercice 12

Définir une fonction `croise_coordonnees : ('a * 'b) list -> ('c * 'd) list -> ('a * 'b * 'c * 'd) list` telle que `croise_coordonnees a b` retourne tous les 4-uplets qu'il est possible de former avec un élément de a et un élément de b. Par exemple,

```
# croise_coordonnees  [(1,2);(3,4)] [(5,6);(7,8)];;
- : (int * int * int * int) list =
[(1, 2, 5, 6); (1, 2, 7, 8); (3, 4, 5, 6); (3, 4, 7, 8)]
```

Exercice 13

Définir une fonction `coordonnees_rectangles : int -> (int * int * int * int) list` telle que `coordonnees_rectangles n` retourne la liste des coordonnées de tous les rectangles qu'il est possible de former à l'intérieur d'un carré de côté n. Par exemple,

```
# coordonnees_rectangles 2;;
- : (int * int * int * int) list =
[(1, 1, 1, 1); (1, 1, 1, 2); (1, 1, 2, 2); (1, 2, 1, 1); (1, 2, 1, 2);
 (1, 2, 2, 2); (2, 2, 1, 1); (2, 2, 1, 2); (2, 2, 2, 2)]
```

Exercice 14

Définir une fonction `tous_les_rectangles : int -> string list` telle que `tous_les_rectangles n` retourne une liste contenant les représentations sous forme de chaînes de caractères de tous les rectangles qu'il est possible de former à l'intérieur d'un carré de côté n. Par exemple,

```
# tous_les_rectangles 2;;
- : string list =
["* . \n. . \n\n"; "* * \n. . \n\n"; ". * \n. . \n\n"; "* . \n* . \n\n";
 "* * \n* * \n\n"; ". * \n. * \n\n"; ". . \n* . \n\n"; ". . \n* * \n\n";
 ". . \n. * \n\n"]
```

Exercice 15

Combien peut-on former de rectangles d'étoiles dans un carré de côté n ?

Chapitre 3

Autour du programme de maths

3.1 Calcul matriciel

3.1.1 Représentation des matrices

Dans cette section, nous mettrons au point des fonctions permettant de créer des matrices et de les afficher sous un format lisible.

Exercice 1

Définir la fonction `make_matrix : int -> int -> float array array` telle que `make_matrix n p` retourne un tableau de n tableaux à p éléments contenant des 0., par exemple :

```
# make_matrix 3 4;;
- : float array array =
[|[|0.; 0.; 0.; 0.|]; [|0.; 0.; 0.; 0.|]; [|0.; 0.; 0.; 0.|]|]
```

Notez bien qu'en caml light, `Array` est remplacé par `vect`.

Exercice 2

Définir les fonctions `nb_lines : 'a array -> int` et `nb_columns : 'a array array -> int` retournant respectivement le nombre de lignes et de colonnes de la matrice passée en paramètre.

Exercice 3

Définir la fonction `fill_matrix : 'a array array -> (int * int -> 'a) -> unit` telle que `fill_matrix m f` place dans la matrice m les images de ses coordonnées par f, ainsi on aura $m_{ij} = f(i, j)$. Par exemple :

```
# let m = make_matrix 3 4;;
val m : float array array =
  [|[|0.; 0.; 0.; 0.|]; [|0.; 0.; 0.; 0.|]; [|0.; 0.; 0.; 0.|]|]
# fill_matrix m (function i,j -> float_of_int (i + j + 1));;
- : unit = ()
# m;;
- : float array array =
[|[|1.; 2.; 3.; 4.|]; [|2.; 3.; 4.; 5.|]; [|3.; 4.; 5.; 6.|]|]
```

Exercice 4

Définir la fonction `identity : int -> float array array` telle que `identity n` retourne la matrice identité d'ordre n.

Exercice 5

Définir la fonction `copy_matrix : float array array -> float array array` telle que `copy_matrix` retourne une copie de la matrice m.

Exercice 6

Définir la fonction `integers : int -> float array array` telle que `integers n` retourne la matrice ligne $(123\dots n)$.

Exercice 7

Définir la fonction `str_of_matrix : float array array -> string` retournant une représentation en chaîne de caractères de la matrice passée en paramètre. Par exemple :

```
# print_string (str_of_matrix (identity 4));;
 1. 0. 0. 0.
 0. 1. 0. 0.
 0. 0. 1. 0.
 0. 0. 0. 1.
- : unit = ()
```

Exercice 8

Définir la fonction `print_matrix : float array array -> unit` affichant la matrice passée en paramètre. Par exemple :

```
# print_matrix (identity 4);;
 1. 0. 0. 0.
 0. 1. 0. 0.
 0. 0. 1. 0.
 0. 0. 0. 1.
- : unit = ()
```

Exercice 9

Définir la fonction `matrix_of_list : float list -> float array array` retournant une matrice ligne contenant les éléments de la liste passée en paramètre. Par exemple,

```
# matrix_of_list [1.;2.;4.;3.;5.;6.];;
- : float array array = [|[|1.; 2.; 4.; 3.; 5.; 6.|]|]
```

Exercice 10

Définir la fonction `list_of_vector : 'a array -> 'a list` retournant une liste contenant les éléments de la matrice passée en paramètre. Par exemple,

```
# list_of_vector  [|1;2;3|];;
- : int list = [1; 2; 3]
```

Exercice 11

Définir la fonction `list_of_matrix : 'a array array -> 'a list list` retournant une liste de liste contenant les éléments de la matrice passée en paramètre. Par exemple,

```
# list_of_matrix [|[|1;2;3|]; [|4;5;6|]|];;
- : int list list = [[1; 2; 3]; [4; 5; 6]]
```

3.1.2 Opérations matricielles

Nous allons maintenant implémenter les opérations les plus courantes sur les matrices : somme, produit, transposition, etc.

Exercice 12

Définir la fonction `transpose : float array array -> float array array` retournant la transposée de la matrice passée en paramètre.

Exercice 13

Définir la fonction `rotation : float array array -> float array array` retournant une copie de la matrice passée en paramètre transformée de la façon suivante :

```
# rotation [|[|1. ; 2. ; 3.|] ; [|4. ; 5. ; 6. |]; [|7. ; 8. ; 9. |]; [|10. ; 11. ; 12. |]|];;
- : float array array =
[|[|12.; 11.; 10.|]; [|9.; 8.; 7.|]; [|6.; 5.; 4.|]; [|3.; 2.; 1.|]|]
```

Exercice 14

Définir la fonction `add_matrix : float array array -> float array array -> float array array` retournant la somme des matrices passées en paramètre. Vous créerez l'exception `Dimensions_mismatch` et l'utiliserez dans cette fonction si les deux matrices ne peuvent être additionnées..

Exercice 15

Définir la fonction `trace : float array array -> float` retournant la trace de la matrice passée en paramètre. Vous lèverez l'exception `Dimensions_mismatch` si la matrice n'est pas carrée.

Exercice 16

Définir la fonction `mult_matrix_const : float array array -> float -> float array array` telle que `mult_matrix_const m k` retourne le produit de la matrice m par la constante k.

Exercice 17

Définir la fonction `mult_matrix : float array array -> float array array -> float array array` retournant le produit des matrices passées en paramètre. Vous contrôlerez les dimensions des deux matrices.

Exercice 18

Définir la fonction `exp_matrix : float array array -> int -> float array array` telle que `exp_matrix m n` retourne m élevée à la puissance n.

3.1.3 Inversion de matrices

Nous allons maintenant nous intéresser à une opération davantage subtile, l'inversion de matrice. Bien qu'il existe de nombreux algorithmes très performants pour ce faire, nous allons implémenter celui que vous connaissez déjà. Pour inverser une matrice m, on appliquera le pivot de Gauss sur m jusqu'à obtenir la matrice identité, on appliquant simultanément ces opérations sur la matrice identité, on obtiendra à la fin l'inverse de m.

Pour des raisons de simplicité, les indices que nous utiliserons ici ne seront pas les indices mathématiques (commençant à 1), mais les indices telles qu'ils sont utilisés en caml, en comptant à partir de 0.

Exercice 19

Définir la fonction `add_linear_combination : float array array -> int * float -> int * float -> unit` telle que `add_linear_combination m (lg, wg) (ld, wd)` effectue l'opération $L_{lg} \leftarrow wgL_{lg} + wdL_{ld}$ sur la matrice m. Cette opération consiste à remplacer la ligne d'indice lg par la combinaison linéaire obtenue en additionnant les lignes d'indices lg et ld pondérées par les réels wd et wd.

Exercice 20

Définir la fonction `pivot : float array array -> float array array -> int * int -> unit`. `pivot m inv (a, b)` effectue sur m une itération de l'algorithme du pivot de Gauss en utilisant m_{ab} comme pivot. Les 0 se trouvant dans les $b - 1$ premières colonnes de m ne doivent pas être éliminés par l'exécution de la fonction, et les coefficients de trouvant en dessous du pivot doivent passer à 0.

La matrice m ne doit pas être recopiée mais modifiée, les opérations effectuées sur les lignes de m doivent être répercutées sur la matrice inv.

Par exemple, l'appel de `pivot m inv (0, 0)` sur les matrices

$$m = \begin{pmatrix} 2 & 5 & 8 \\ 4 & 7 & 0 \\ 1 & 9 & 3 \end{pmatrix} \text{ et } inv = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

transforme les variable m et inv de la façon suivante :

$$m = \begin{pmatrix} 2 & 5 & 8 \\ 0 & -6 & 32 \\ 0 & 13 & -2 \end{pmatrix} \text{ et } inv = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 2 & 0 \\ -1 & 0 & 2 \end{pmatrix}$$

Ce qui donne

```
val m : float array array =
  [|[|2.; 5.; 8.|]; [|4.; 7.; 0.|]; [|1.; 9.; 3.|]|]
val inv : float array array =
  [|[|1.; 0.; 0.|]; [|0.; 1.; 0.|]; [|0.; 0.; 1.|]|]
# pivot m inv (0, 0);;
- : unit = ()
# m;;
- : float array array =
[|[|2.; 5.; 8.|]; [|0.; -6.; -32.|]; [|0.; 13.; -2.|]|]
# inv;;
- : float array array = [|[|1.; 0.; 0.|]; [|-4.; 2.; 0.|]; [|-1.; 0.; 2.|]|]
```

Exercice 21

Il n'est possible d'utiliser m_{ab} comme pivot que s'il est est non nul. Si $m_{ab} = 0$, il convient de permuter la ligne d'indice a avec une ligne k (vérifiant $k > a$) de sorte que le pivot soit non nul. Définir la fonction `find_pivot : float array array -> int -> int` telle que `find_pivot m i` recherche un pivot dans la colonne d'indice i, et retourne l'indice de la ligne k telle que $m_{ki} \neq 0$ et $k \geq i$. Par exemple,

```
# find_pivot [|[|0.; 5.; 8.|]; [|0.; 7.; 0.|]; [|1.; 9.; 3.|]|] 0;;
- : int = 2
```

Si un tel k n'existe pas, alors la matrice n'est pas inversible, et vous lèverez l'exception `Singular_matrix`.

Exercice 22

Définir la fonction `swap_items : 'a array array -> int * int -> int * int -> unit` telle que `swap_items m (l1, c1) (l2, c2)` permute dans la matrice m les éléments $m_{l1,c1}$ et $m_{l2,c2}$.

Exercice 23

Définir la fonction `swap_lines : 'a array array -> int -> int -> unit` telle que `swap_lines m i k` permute dans la matrice m les lignes d'indices i et k.

Exercice 24

Définir la fonction `iteration_pivot : float array array -> float array array -> int -> unit` telle que `iteration_pivot m inv i` recherche dans la colonne i de la matrice m un pivot, l'amène en m_{ii} (en permutant éventuellement la ligne i avec une autre ligne), et annule tous les m_{ij} $(i < j)$ avec des opérations sur les lignes de m.

Toutes les opérations sur les lignes de m seront répercutées sur la matrice inv.

Exercice 25

Définir la fonction `reduce_diagonal : float array array -> float array array -> unit` telle que si m est une matrice diagonale, alors `reduce_diagonal m inv` amène tous les coefficients diagonaux à 1 avec des opérations sur les lignes de m. Toutes ces opérations sur les lignes de m doivent être répercutées sur inv.

Exercice 26

Définir la fonction `apply_function : ('a -> 'a) -> 'a -> int -> 'a` telle que `apply_function f x n` retourne l'image de x par la fonction obtenue en composant f avec elle-même n fois. Par exemple, `apply_function f x 3` retourne $f^3(x) = f(f(f(x)))$. On considérera que f^0 est la fonction identité.

Exercice 27

Définir la fonction `inverse_matrice : float array array -> float array array` telle que `inverse_matrice m` retourne la matrice inverse de m. Attention, `inverse_matrice` ne doit pas produire d'effet de bord.

Exercice 28

Définir la fonction `solve_system : float array array -> float array array -> float array array` tel que `solve_system a y` retourne la solution x de l'équation $Ax = y$.

3.2 Polynômes et analyse numérique

3.2.1 Représentation des polynômes

Nous souhaitons représenter des polynômes à variable réelle et à coefficients réels. Nous représenterons un monôme par un couple (c, e), où c est le coefficient du monôme de degré e. Un polynôme sera représenté par une liste de couples disposés par ordre de degrés strictement décroissant. Par exemple, le polynôme $x^2 - x + 1$ est représenté par la liste `[(2.,2) ;(-1., 1) ; (1.,0)]`.

Exercice 1

Comment représenter le polynôme $x + 2 - x^3$?

Exercice 2

Quel polynôme est représenté par `[(4., 4) ; (2., 3) ; (-3., 2) ; (3., 0)]` ?

Exercice 3

Définir la fonction `polynomeUnite : (float * int) list` telle que `polynomeUnite ;;` retourne le polynome unité, c'est-à-dire l'élément neutre pour la multiplication dans l'anneau des polynômes.

Exercice 4

Définir la fonction `polynomeNul : (float * int) list` telle que `polynomeNul ;;` retourne le polynome nul, c'est-à-dire l'élément neutre pour l'addition dans l'anneau des polynômes.

Exercice 5

Définir la fonction `strOfMonome : float * int -> string` telle que `strOfMonome m` retourne une représentation au format chaîne de caractères du monôme m.

```
# strOfMonome (4., 3);;
- : string = "4.*X^3"
# strOfMonome (-2., 1);;
- : string = "-2.*X^1"
# strOfMonome (2., 0);;
- : string = "2.*X^0"
```

Exercice 6

Définir la fonction `strOfPolynome : (float * int) list -> string` telle que `strOfPolynome p` retourne une représentation au format chaîne de caractères du polynôme p. Par exemple,

```
# strOfPolynome [(0., 3) ; (-1., 2); (0., 1) ; (2., 0)];;
- : string = "0.*X^3 + -1.*X^2 + 0.*X^1 + 2.*X^0"
```

Exercice 7

Définir la fonction `expOfPolynome : (float * int) list -> exp` telle que `expOfPolynome p` convertit le polynôme p en expression de la forme donnée dans 1.3.2, par exemple

```
# let k = expOfPolynome [(-1., 2);(2., 0)];;
val k : exp =
  Somme (Produit (Const (-1.), Puiss (X, 2)),
   Somme (Produit (Const 2., Puiss (X, 0)), Const 0.))
# str_of_exp k;;
- : string = "((-1.*(x^2))+((2.*(x^0))+0.))"
```

3.2.2 Opérations sur les polynômes

Nous allons dans cette section implémenter les opérations les plus courantes sur les polynômes : addition, soustraction et produit. D'autres opérations, plus élaborées, seront examinées dans les sections suivantes.

Pour toutes les questions suivantes, un coefficient sera considéré comme nul s'il contient une valeur inférieure à 10^{-15}.

Exercice 8

Définir la fonction `absFloat : float -> float` telle que `absFloat x` retourne la valeur absolue de x.

Exercice 9

Définir la fonction `estNul : float -> bool` telle que `estNul x` retourne vrai si et seulement si $|x| \leq 10^{-15}$.

Exercice 10

Définir la fonction `addPolynomes : (float * 'a) list -> (float * 'a) list -> (float * 'a) list` telle que `addPolynomes a b` retourne la somme des polynômes a et b.

Exercice 11

Définir la fonction `multConst : (float * 'a) list -> float -> (float * 'a) list` telle que `multConst p k` retourne le produit du polynôme p par la constante k.

Exercice 12

Définir la fonction `subPolynomes : (float * 'a) list -> (float * 'a) list -> (float * 'a) list` telle que `subPolynomes a b` retourne le résultat de la soustraction du polynôme b au polynôme a.

Exercice 13

Définir la fonction `multXk : ('a * int) list -> int -> ('a * int) list` telle que `multXk p k` retourne le polynôme p multiplié par X^k.

Exercice 14

Définir la fonction `multMonome : (float * int) list -> float * int -> (float * int) list` telle que `multMonome p (c, e)` retourne le produit de p par cX^e.

Exercice 15

Définir la fonction `multPolynomes : (float * int) list -> (float * int) list -> (float * int) list` telle que `multPolynomes p q` retourne le produit des polynômes p et q.

Exercice 16

Définir la fonction `expPolynome : (float * int) list -> int -> (float * int) list` telle que `expPolynome p n` retourne le polynôme p élevé à la puissance n.

Exercice 17

Définir la fonction `polynomeOfExp : exp -> (float * int) list` telle que `polynomeOfExp e` convertit en polynôme l'expression e donnée dans la forme décrite en 1.3.2, par exemple

```
# let k = polynomeOfExp (Puiss(Somme(Const(1.), X), 4));;
val k : (float * int) list = [(1., 4); (4., 3); (6., 2); (4., 1); (1., 0)]
# strOfPolynome k;;
- : string = "1.*X^4 + 4.*X^3 + 6.*X^2 + 4.*X^1 + 1.*X^0"
```

3.2.3 Évaluation et méthode de Horner

Exercice 18

Définir la fonction `floatExp : float -> int -> float` telle que `floatExp x n` retourne x^n.

Exercice 19

Définir la fonction `evalueMoche : (float * int) list -> float -> float` telle que `evalueMoche p x` retourne l'image de x par la fonction polynôme représentée par p.

Exercice 20

Quelle la complexité de l'évaluation d'un polynôme de degré n avec la fonction `evalueMoche` ?

Exercice 21

Nous allons redéfinir la fonction d'évaluation d'un polynôme en utilisant la méthode de Horner. A titre d'exemple, observons que le polynôme $5x^3+x^2+3x-1 = (5x^2+x+3)x-1$, que $5x^2+x+3 = (5x+1)x+3$, puis que $5x+1 = (5)x+1$, ce qui nous donne

$$5x^3 + x^2 + 3x - 1 = (((5)x + 1)x + 3)x - 1$$

On généralise ce procédé avec la relation de récurrence

$$a_n x^n + \ldots + a_1 x + a_0 = (a_n x^{n-1} + \ldots + a_1)x + a_0$$

Utiliser ce procédé pour reformuler les expressions

$$x^3 - 2x^2 + 7x - 1$$

et

$$4x^7 + 2x^3 - 2x^2 - x + 1$$

Exercice 22

Définir la fonction `evalHorner : (float * int) list -> float -> float` telle que `evalHorner p x` retourne l'image de x par la fonction polynôme représentée par p. Vous utiliserez une fonction auxiliaire avec un accumulateur.

Exercice 23

Quelle la complexité de l'évaluation d'un polynôme de degré n avec la fonction `evalHorner` ?

3.2.4 Dérivée et intégrale

Exercice 24

Définir la fonction `derivePolynome : (float * int) list -> (float * int) list` telle que `derivePolynome p` retourne la dérivée de p.

Exercice 25

Définir la fonction `primitivePolynome : (float * int) list -> (float * int) list` telle que `primitivePolynome p` retourne une primitive de p.

Exercice 26

Définir la fonction `integrePolynome : (float * int) list -> float -> float -> float` telle que `integrePolynome p a b` retourne $\int_a^b p(x)dx$.

3.2.5 Calculs approchés d'intégrales

Lorsque les fonctions à intégrer sont difficiles à primitiver, il est usuel d'utiliser des algorithmes donnant rapidement des valeurs approchées d'une précision plus ou moins convenable selon le domaine d'application. Nous implémenterons la méthode des rectangles ainsi que la méthode des trapèzes.

Dans les deux méthodes que nous implémenterons, nous utiliserons la subdivision du segment $[a, b]$ en n segments de même taille, que nous noterons $[a_0, a_1], [a_1, a_2], \ldots, [a_n, a_{n-1}]$ et tels que $\forall i \in \{1, \ldots, n\}$ les valeurs $(a_i - a_{i-1})$ soient identiques.

Exercice 27

La méthode des rectangles consiste à approcher l'aire $\int_{a_{i-1}}^{a_i} p(x)dx$ par la surface du rectangle délimité par les points de coordonnées $(a_{i-1}, 0)$, $(a_i, 0)$, $(a_{i-1}, p(a_{i-1}))$ et $(a_i, p(a_{i-1}))$. L'intégrale $\int_a^b p(x)dx$ sera donc approchée par la somme sur i des $\int_{a_{i-1}}^{a_i} p(x)dx$.

Définir la fonction `rectangles : (float * int) list -> float -> float -> int -> float` telle que `rectangles p a b n` retourne une approximation de $\int_a^b p(x)dx$ en utilisant n rectangles.

Exercice 28

La méthode des trapèzes est une variante dans laquelle on approche chaque $\int_{a_{i-1}}^{a_i} p(x)dx$ par la surface du trapèze délimité par les points de coordonnées $(a_{i-1}, 0)$, $(a_i, 0)$, $(a_{i-1}, p(a_{i-1}))$ et $(a_i, p(a_i))$.

Définir la fonction `trapezes : (float * int) list -> float -> float -> int -> float` telle que `trapezes p a b n` retourne une approximation de $\int_a^b p(x)dx$ en utilisant n trapèzes. Il est conseillé de chercher à simplifier la formule donnant la valeur de l'approximation et de la calculer avec une boucle.

3.2.6 Interpolation

A partir d'une liste de n points du plan $[(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)]$, nous souhaitons déterminer un polynôme p, de degré n tel que tel que $\forall i \in \{0, \ldots, n\}$, $p(x_i) = y_i$.

Exercice 29

Définir la fonction `facteurLagrange : float -> (float * int) list` telle que `facteurLagrange z` retourne le polynôme $(x - z)$.

Exercice 30

Étant donné i, posons

$$f_i(x) = \Pi_{j \neq i}(x - x_j)$$

Exercice 31

Définir la fonction `lagrange_f_i : (float * 'a) list -> float -> (float * int) list` telle que si *pts* est la liste de points $[(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)]$, `lagrange_f_i pts x_i` retourne f_i.

Exercice 32

Déterminer en fonction de f_i un polynôme p_i vérifiant $p_i(x_i) = y_i$ et $\forall j\{0, \ldots, n\}$ tel que $i \neq j$, $p_i(x_j) = 0$.

Exercice 33

Définir la fonction `lagrange_p_i : (float * 'a) list -> float * float -> (float * int) list` telle que `lagrange_p_i pts (x_i, y_i)` retourne le polynôme p_i.

Exercice 34

Déterminer en fonction des p_i un polynôme p interpolant les points $[(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)]$.

Exercice 35

Définir la fonction `interpolationLagrange : (float * float) list -> (float * int) list` telle que `interpolationLagrange pts` retourne un polynôme interpolant les points *pts*.

Exercice 36

Il existe une façon assez élégante d'interpoler un polynôme $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ passant par les points $[(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)]$ en posant l'équation matricielle

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^j & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^j & \ldots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \\ 1 & x_i & x_i^2 & \ldots & x_i^j & \ldots & x_i^n \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^j & \ldots & x_n^n \end{pmatrix} \times \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_i \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_i \\ \vdots \\ y_n \end{pmatrix}$$

Définir la fonction `interpolationVandermonde : (float * float) list -> (float * int) list` telle que`interpolationVandermonde pts` retourne un polynôme interpolant les points *pts*. Vous utiliserez l'algorithme de résolution de systèmes linéaires codé dans le TP précédent.

3.2.7 Méthode de Newton

La méthode de Newton consiste, étant donnée une fonction dérivable f, à définir une suite numérique convergeant vers un point l solution de l'équation $f(l) = 0$. Les conditions garantissant la convergence sortent du cadre de cet exercice.

Étant donné un point x_n de cette suite, on détermine le point suivant x_{n+1} en calculant la tangente T à f au point d'abscisse x_n. x_{n+1} est l'abscisse de l'intersection de T et de l'axe des abscisses.

Exercice 37

Exprimer x_{n+1} en fonction de f, de f' et de x_n.

Exercice 38

Définir la fonction `newton : (float * int) list -> float -> int -> float` telle que si f est la fonction polynôme représentée par p, et si le premier point de la suite x est z, alors `newton p z n` retourne le n-ème point de la suite x.

3.2.8 Division euclidienne

3.2.9 Méthode de Sturm

Chapitre 4

Quelques corrigés

4.1 Initiation

```
(* 1.1.1 factorielle *)

(* exercice 1 *)

let rec factorielle = function
        0 -> 1
        | n -> n * (factorielle (n-1));;

(* exercice 2 *)

let rec factorielle_acc n k =
        if n = 0 then
                k
        else
                factorielle_acc (n-1) (n*k);;

(* exercice 3 *)

let factorielle_bis n = factorielle_acc n 1;;

(* 1.1.2 exponentiation lente *)

(* exercice 4 *)

let rec slow_exp b = function
    0 -> 1
  | n -> slow_exp b (n - 1) * b;;

(* exercice 5 *)

let rec slow_exp_acc b n acc =
  if (n = 0) then
    acc
  else
    slow_exp_acc b (n - 1) (acc * b);;

(* exercice 6 *)

let slow_exp_bis b n = slow_exp_acc b n 1;;

(* 1.1.3 exponentiation rapide *)

(* exercice 7 *)

let rec even_rec = function
  0 -> true
  | 1 -> false
  | n -> even_rec (n - 2);;

let even n = n mod 2 = 0;;

(* exercice 8 *)

let rec fast_exp b n =
  if (n = 0) then
    1
```

```
  else
    if (even n) then
      fast_exp (b*b) (n/2)
        else
      fast_exp (b) (n-1) * b;;

(* exercice 9 *)

(*
  Le parametre n de la fonction decrit une suite d'entiers strictement
  decroissante au fil des appels recursifs. Donc la fonction se termine.

  Notez bien que ca ne serait pas le cas si l'on avait place
        fast_exp fast_exp (b) (n/2) 2
  dans le premier appel recursif.
*)

(* exercice 10 *)

let rec fast_exp_acc b n acc =
  if (n = 0) then
    acc
  else
    if (even n) then
      fast_exp_acc (b*b) (n/2) acc
        else
      fast_exp_acc (b) (n-1) (acc * b);;

let fast_exp_bis b n = fast_exp_acc b n 1;;

(* 1.1.4 pgcd *)

(* exercice 11 *)

(*
  Lors de chaque appel recursif Le deuxieme parametre de la fonction (m)
  devient (n mod m). Or |n mod m| < |m|, ce qui s'obtient aisement avec la
  definition de la division dans z :

  on divise n par m en determinant q et r tels que n = mq + r, |r| < |m|

  Ici on a r = n mod m. Comme le deuxieme parametre est toujours strictement
  decroissant, et que l'algorithme s'arrete lorsqu'il atteint 0, la fonction
  se termine.
*)

(* exercice 12 *)

let rec gcd n m =
  if (m = 0) then
    n
  else
    gcd m (n mod m);;

(* 1.1.5 alphabet *)

(* exercice 13 *)

let lettre_suivante a = char_of_int (int_of_char a + 1);;

(* exercice 14 *)

let figure_of_int n = char_of_int (n - 1 + (int_of_char '1'));;

(* exercice 15 *)

let compare_lettres a b = int_of_char a <= int_of_char b;;

(* exercice 16 *)

let rec alphabet lettre_depart lettre_arrivee =
  if (compare_lettres lettre_depart lettre_arrivee) then
    (
     print_char lettre_depart;
     alphabet (lettre_suivante lettre_depart) lettre_arrivee
    )
  else
    ();;

(* 1.1.6 mise sous forme recursive *)
```

```
(* exercice 17 *)

  let rec mult x y =
    if (y = 0) then
      y
    else
      mult x (y - 1) + x;;

(* exercice 18 *)

let rec mult_acc x y acc =
  if (y = 0)
  then acc
  else
    mult_acc x (y - 1) (acc + x);;

let mult_bis x y = mult_acc x y 0;;

(* 1.1.7 indicatrice d'Euler *)

(* exercice 19 *)

let rec nb_prem a n =
  if (a >= n) then
    0
  else
    (
     if (gcd a n = 1) then
       1
     else
       0
    )
     + nb_prem (a + 1) n;;

let ind_euler n = nb_prem 0 n;;
```

4.2 Listes

```
(* exercice 1 *)

let rec n_a_un = function
    0 -> []
  | n -> n::(n_a_un (n - 1));;

(* exercice 2 *)

let rec un_a_n_acc n acc =
  if (n = 0) then
    acc
  else
    un_a_n_acc (n-1) (n::acc);;

let un_a_n n = un_a_n_acc n [];;

(* exercice 3 *)

let rec supprime l n =
  if (n <= 0) then
    l
  else
    match l with
      [] -> []
    | head::tail -> supprime tail (n-1);;

(* exercice 4 *)

let rec long_prefixe l =
  match l with
    [] -> 0
  | x::[] -> 1
  | x::y::tail ->
      if (x = y) then
        1 + long_prefixe (y::tail)
      else
        1;;
```

4.3 Types récursifs

```ocaml
(* 1.3.1 Tri par arbre binaire de recherche *)

type 'a bin_tree = Empty | Node of 'a bin_tree * 'a * 'a bin_tree;;

(* exercice 1 *)

let rec ins_abr f x t = match t with
  Empty -> Node (Empty, x , Empty)
| Node(left, y, right) -> if (f x y) then
    Node(ins_abr f x left, y, right)
      else
    Node(left, y, ins_abr f x right);;

(* exercice 2 *)

let rec abr_of_list f l = match l with
  [] -> Empty
| x::tail -> ins_abr f x (abr_of_list f tail);;

(* exercice 3 *)
let rec list_of_abr = function
    Empty -> []
  | Node(left, x, right) -> (list_of_abr left)@[x]@(list_of_abr right);;

(* exercice 4 *)

let tri_liste f l = list_of_abr (abr_of_list f l);;

(* 1.3.2 Fonctions polynomes *)

type exp =
    Const of float
  | X
  | Somme of exp * exp
  | Produit of exp * exp
  | Puiss of exp * int;;

(* exercice 5 *)

let rec str_of_exp = function
    Const(x) -> (string_of_float x)
  | X -> "x"
  | Somme(left, right) -> "("^(str_of_exp left)^ "+"^(str_of_exp right)^ ")"
  | Produit(left, right) -> "("^(str_of_exp left)^ "*"^(str_of_exp right)^ ")"
  | Puiss(left, n) -> "("^(str_of_exp left)^ "^"^(string_of_int n)^ ")" ;;

(* exercice 6 *)

let rec image e x = match e with
    Const(c) -> c
  | X -> x
  | Somme(left, right) -> image left x +. image right x
  | Produit(left, right) -> image left x *. image right x
  | Puiss(left, n) -> let rec fast_exp b = function
        0 -> 1.
      | n -> if (n mod 2 = 0) then
          fast_exp (b*.b) (n/2)
      else
          fast_exp (b*.b) ((n-1)/2) *. b
        in
    fast_exp (image left x) n;;

(* exercice 7 *)

let rec derive  = function
    Const(_) -> Const(0.)
  | X -> Const(1.)
  | Somme(left, right) -> Somme(derive left, derive right)
  | Produit(left, right) -> Somme(Produit(derive left, right), Produit(left, derive right))
  | Puiss(left, n) -> Produit(Produit(Const(float_of_int n), derive left), Puiss(left, n-1));;

(* exercice 8 *)

let rec degre = function
    Const(_) -> 0
  | X -> 1
  | Somme(left, right) -> max (degre left) (degre right)
  | Produit(left, right) -> degre left + (degre right)
  | Puiss(left, n) -> degre left * n;;
```

4.4 Le carré qui rend fou

```
(* Affichage de carrÃ©s *)

(* exercice 1 *)

let rec ligne n =
  ". " ^ (
          if (n > 1) then
            ligne (n - 1)
          else
            "\n"
         );;

(* exercice 2 *)

let rec carre n =
  let rec carre_aux = function
      0 -> "\n"
    | k -> ligne n ^ (carre_aux (k - 1))
  in
  carre_aux n;;

(* Affichage de carrÃ©s d'Ã©toiles*)

(* exercice 3 *)

let rec etoiles a b n =
  if (n = 0) then
    "\n"
  else
    (
    if (a <= 1 && b >= 1) then
      "* "
    else
      ". "
    )
    ^ (etoiles (a-1) (b-1) (n-1));;

(* exercice 4 *)

let carre_etoiles (lhaut, lbas, cgauche, cdroite) n =
  let rec carre_aux lhaut lbas k =
    if (k > n) then
      "\n"
    else
      (
       (
        if (lhaut <= 1 && lbas >= 1) then
          etoiles cgauche cdroite n
        else
          ligne n
       )
       ^ (carre_aux (lhaut - 1) (lbas - 1) (k + 1))
      )
  in
  carre_aux lhaut lbas 1;;

(* EnumÃ©ration des carrÃ©s  *)

(* exercice 5 *)

let rec couples_coordonnees_a_fixe a b =
  if (a > b) then
    []
  else
    (couples_coordonnees_a_fixe a (b-1))@[(a, b)];;

(* exercice 6 *)

let couples_coordonnees n =
  let rec couples_coordonnees_aux a =
    if (a > n) then
      []
    else
      couples_coordonnees_a_fixe a n@(couples_coordonnees_aux (a+1))
  in
  couples_coordonnees_aux 1;;

(* exercice 7 *)
```

```
let rec coordonnees_carres_colonnes_fixees (cgauche, cdroite) n =
  if (n <= cdroite - cgauche) then
    []
  else
    coordonnees_carres_colonnes_fixees (cgauche, cdroite) (n-1)
    @[(n - (cdroite - cgauche), n, cgauche, cdroite)];;

(* exercice 8 *)

let coordonnees_carres n =
  let colonnes = couples_coordonnees n in
  let rec complete_coordonnees colonnes =
    match colonnes with
      [] -> []
    | h::t -> coordonnees_carres_colonnes_fixees h n
          @(complete_coordonnees t) in
  complete_coordonnees colonnes;;

(* exercice 9 *)

let tous_les_carres n =
  let rec liste_carres  l = match l with
    [] -> []
  | coord::t -> carre_etoiles coord n::(liste_carres t)
  in
  liste_carres (coordonnees_carres n);;

(* exercice 10 *)

let rec print_str_list = function
    [] -> print_string "\n";
  | h::t -> print_string h; print_str_list t;;

print_str_list (tous_les_carres 3);;

(* EnumÃ©ration des rectangles *)

(* exercice 11 *)

let rec concat_produits (a, b) l = match l with
  [] -> []
| (c, d)::t -> (a, b, c, d)::(concat_produits (a, b) t);;

(* exercice 12 *)

let rec croise_coordonnees a b = match a with
  [] -> []
| h::t -> (concat_produits h b)@(croise_coordonnees t b);;

(* exercice 13 *)

let coordonnees_rectangles n =
  let colonnes = couples_coordonnees n in
  croise_coordonnees colonnes colonnes;;

(* exercice 14 *)

let tous_les_rectangles n =
  let rec liste_rectangles  l = match l with
    [] -> []
  | coord::t -> carre_etoiles coord n::(liste_rectangles t)
  in
  liste_rectangles (coordonnees_rectangles n);;

(* exercice 15 *)

(*

Il existe C n+1 n faÃ§ons de choisir les lignes et autant de faÃ§ons de
choisir les colonnes, donc...

*)
```

4.5 Algèbre

```
let make_vect = Array.make;;
let length_vect = Array.length;;
```

```
(**********************************************************************)

let make_matrix n p =
  let m = make_vect n [||] in
  for i = 0 to (n-1) do
    m.(i) <- make_vect p 0.;
  done;
  m;;

(**********************************************************************)

let nb_lines m = length_vect m;;

(**********************************************************************)

let nb_columns m = length_vect m.(0);;

(**********************************************************************)

let fill_line m i f =
  let p = nb_columns m in
  for j = 0 to (p-1) do
    m.(i).(j) <- f j;
  done;;

(**********************************************************************)

let fill_matrix m f =
  let n = nb_lines m in
  for i = 0 to (n - 1) do
      fill_line m i (function j -> f (i,j));
  done;;

(**********************************************************************)

let identity n =
  let m = make_matrix n n in
  fill_matrix m (function i,j -> if (i = j) then 1. else 0.);
  m;;

(**********************************************************************)

let copy_matrix m =
  let c = make_matrix (nb_lines m) (nb_columns m) in
  fill_matrix c (function i,j -> m.(i).(j));
  c;;

(**********************************************************************)

 let integers n =
  let m = make_matrix 1 n in
  fill_matrix m (function _,j -> float_of_int (j+1));
  m;;

(**********************************************************************)

let str_of_matrix m =
  let str = ref "" in
  let n = nb_lines m in
  let p = nb_columns m in
  for i = 0 to (n - 1) do
      for j = 0 to (p - 1) do
        str := !str^" "^string_of_float m.(i).(j);
      done;
    str := !str^"\n"
  done;
  !str;;

(**********************************************************************)

let print_matrix m =
  print_string (str_of_matrix m);;

(**********************************************************************)

let matrix_of_list l =
  let m = make_matrix 1 (List.length l) in
  let rec fill i l = match l with
      [] -> ()
    | h::t -> m.(0).(i) <- h ; fill (i+1) t in
  fill 0 l;
```

```
  m;;

(**************************************************************)

let list_of_vector v =
  let rec list_of_vector_acc acc = function
      -1 -> acc
    | j -> list_of_vector_acc (v.(j)::acc) (j-1) in
  list_of_vector_acc [] (length_vect v - 1);;

(**************************************************************)

let list_of_matrix m =
  let rec list_of_matrix_acc acc m = function
      -1 -> acc
    | i -> list_of_matrix_acc ((list_of_vector m.(i))::acc) m (i-1) in
  list_of_matrix_acc [] m (nb_lines m - 1);;

(**************************************************************)

let transpose m =
  let n = nb_lines m in
  let p = nb_columns m in
  let t = make_matrix p n in
  fill_matrix t (function i,j -> m.(j).(i));
  t;;

(**************************************************************)

let rotation m =
  let n = nb_lines m in
  let p = nb_columns m in
  let t = make_matrix n p in
  fill_matrix t (function i,j -> m.(n - i - 1).(p - j - 1));
  t;;

(**************************************************************)

exception Dimensions_mismatch;;

let add_matrix a b =
  let la = nb_lines a in
  let lb = nb_lines b in
  let ca = nb_columns a in
  let cb = nb_columns b in
  if (la = lb && ca = cb) then
    let c = make_matrix la ca in
    fill_matrix c (function i,j -> a.(i).(j) +. b.(i).(j));
    c;
  else
    raise Dimensions_mismatch;;

(**************************************************************)

let trace m =
  let n = nb_lines m in
  let p = nb_columns m in
  if (n <> p) then
    raise Dimensions_mismatch
  else
    let somme = ref 0. in
    for i = 0 to (n - 1) do
      somme := !somme +. m.(i).(i);
    done;
    !somme;;

(**************************************************************)

let mult_matrix_const m k =
  let n = nb_lines m in
  let p = nb_lines m in
  let c = make_matrix n p in
  fill_matrix c (function (i,j) -> k *. m.(i).(j));
  c;;

(**************************************************************)

let mult_matrix a b =
  let la = nb_lines a in
  let lb = nb_lines b in
  let ca = nb_columns a in
```

```
  let cb = nb_columns b in
  if (ca = lb) then
    let c = make_matrix la cb in
    let fill_function (i,j) =
      (
      let total = ref 0. in
      for k = 0 to (ca - 1) do
        total := !total +. a.(i).(k) *. b.(k).(j);
      done;
      !total;
      ) in
    fill_matrix c fill_function;
    c;
  else
    raise Dimensions_mismatch;;

(*****************************************************************)

let rec exp_matrix m n =
  if (n = 0) then
    identity (nb_columns m)
  else
    mult_matrix m (exp_matrix m (n - 1));;

(*****************************************************************)

let add_linear_combination m (lg, wg) (ld, wd) =
  fill_line m lg (function j -> m.(lg).(j) *. wg +. m.(ld).(j) *. wd);;

(*****************************************************************)

let pivot m inv (a, b) =
  let n = nb_lines m in
  let valeur_pivot = m.(a).(b) in
  for i = (a+1) to (n-1) do
    let r = -.m.(i).(b) in
    add_linear_combination inv (i, valeur_pivot) (a, r);
    add_linear_combination m (i, valeur_pivot) (a, r);
  done;;

(*****************************************************************)

exception Singular_matrix;;

let find_pivot m i =
  let k = ref i in
  let p = nb_lines m in
  while(!k < p && m.(!k).(i) = 0.) do
    k := !k + 1;
  done;
  if !k = p then
    raise Singular_matrix;
  !k;;

(*****************************************************************)

let swap_items m (l1, c1) (l2, c2) =
  let tmp = m.(l1).(c1) in
  m.(l1).(c1) <- m.(l2).(c2);
  m.(l2).(c2) <- tmp;;

(*****************************************************************)

let swap_lines m i k =
  for j = 0 to (nb_columns m - 1) do
    swap_items m (i, j) (k, j);
  done;;

(*****************************************************************)

let iteration_pivot m inv i =
  let k = find_pivot m i in
  if (i <> k) then
    (swap_lines m i k;
     swap_lines inv i k;
    );
  pivot m inv (i, i);;

(*****************************************************************)

let reduce_diagonal m inv =
```

```
  let n = nb_lines m in
  for i = 0 to (n-1) do
    let coeff = m.(i).(i) in
    fill_line m i (function j -> m.(i).(j) /. coeff);
    fill_line inv i (function j -> inv.(i).(j) /. coeff);
  done;;

(*****************************************************************)

let rec apply_function f x = function
    0 -> x
  | n -> apply_function f (f x) (n-1);;

(*****************************************************************)

let invert_matrix m =
  let n = nb_lines m in
  let reduce_and_rotate (a, b) =
    (
     for k = 0 to (n - 2) do
      iteration_pivot a b k;
    done;
     (rotation a, rotation b)
    )
  in
  let (m, inv) = apply_function reduce_and_rotate (copy_matrix m, identity n) 2 in
  reduce_diagonal m inv ;
  inv;;

(*****************************************************************)

let solve_system a y =
  mult_matrix (invert_matrix a) y;;
```

4.6 Polynômes

```
[(-1.,3);(1., 1); (2.,0)];;

(*****************************************************************)

(* 4X^4 + 2X^3 -3X^2 + 3 *)

(*****************************************************************)

let polynomeUnite = [(1., 0)];;
let polynomeNul = [(0., 0)];;

(*****************************************************************)

let strOfMonome (coeff, exp) =
  string_of_float coeff ^ "*X^" ^ string_of_int exp;;

(*****************************************************************)

let rec strOfPolynome = function
  | [] -> strOfPolynome polynomeNul
  | h::[] -> strOfMonome h
  | h::t -> strOfMonome h ^ " + " ^ strOfPolynome t;;

(*****************************************************************)

let rec expOfPolynome = function
    [] -> Const 0.
  | (c, e)::t -> Somme (Produit(Const(c), Puiss(X, e)), expOfPolynome t);;

(*****************************************************************)

let absFloat x =
  if x > 0. then
    x
  else (-.x);;

(*****************************************************************)

let estNul x = absFloat x < 1.e-15;;

(*****************************************************************)

let rec addPolynomes p q = match (p, q) with
| [] , _ -> q
```

```
| _, [] -> p
| (coeffp, expp)::tailp, (coeffq, expq)::tailq ->
    if (expp = expq) then
      (
       if (estNul (coeffp +. coeffq)) then
         (addPolynomes tailp tailq)
      else
         (coeffp +. coeffq, expp)::(addPolynomes tailp tailq)
      )
    else
      if (expp > expq) then
        (coeffp, expp)::(addPolynomes tailp q)
      else
        (coeffq, expq)::(addPolynomes p tailq);;

(*****************************************************************)

  let rec multConst p k = match p with
    [] -> []
  | (c, e)::t ->
      (if (absFloat (k *. c) > 1.e-15) then
        [(k *. c, e)]
      else
        []
      )@(multConst t k);;

(*****************************************************************)

let subPolynomes a b = addPolynomes a (multConst b (-1.));;

(*****************************************************************)

let rec multXk p k = match p with
    [] -> []
  | (c, e)::t -> (c, e+k)::(multXk t k);;

(*****************************************************************)

let multMonome p (c, e) = multXk (multConst p c) e;;

(*****************************************************************)

let rec multPolynomes p q = match p with
[] -> []
| h::t -> addPolynomes (multMonome q h) (multPolynomes t q);;

(*****************************************************************)

let rec expPolynome p = function
    0 -> polynomeUnite
  | n -> multPolynomes p (expPolynome p (n-1));;

(*****************************************************************)

let rec polynomeOfExp = function
    Const(x) -> [(x,0)]
  | X -> [(1.,1)]
  | Somme(left, right) -> addPolynomes (polynomeOfExp left) (polynomeOfExp right)
  | Produit(left, right) -> multPolynomes (polynomeOfExp left) (polynomeOfExp right)
  | Puiss(left, n) -> expPolynome (polynomeOfExp left) n;;

let k = polynomeOfExp (Puiss(Somme(Const(1.), X), 4));;
strOfPolynome k;;

(*****************************************************************)

let rec floatExp x = function
    0 -> 1.
  | n ->
      (if n mod 2 = 0 then
      1.
  else
      x
      ) *. floatExp (x *. x) (n/2);;

(*****************************************************************)

let rec evalueMoche p x = match p with
[] -> 0.
| (c, e)::t -> c *. (floatExp x e) +. evalueMoche t x;;
```

```
(***************************************************************)

(*
un monome de degre k s'evalue en O(log k), qui est le temps necessaire pour
mettre X a la puissance k. On determine le temps que met un polynome
de degre n pour s'evaluer en calculant

u_n = log 1 + log 2 + ... + log n

qui est la somme des temps d'exponentiation de chaqe monome. On a

u_n <= log n + ... + log n <= n log n

evalueMoche s'execute en O(n log n).
*)

(***************************************************************)

(*

((x - 2)x + 7)x -1

(((4x^4 + 2)x - 2)x - 1)x + 1

*)

(***************************************************************)

let rec evalHornerAcc p x acc =
     match p with
       [] -> acc
     | (c, e)::[] -> (acc +. c) *. (floatExp x e)
     | (c, e1)::tail ->
        (match tail with
        | (_, e2)::_ -> evalHornerAcc tail x ((acc +. c) *. (floatExp x (e1 - e2)))
        | _ -> failwith "Oups...");;

let evalHorner p x = evalHornerAcc p x 0. ;;

(***************************************************************)

(*

Supposons que les degres decrive la suite

n, n-1, n-2, ..., 2, 1, 0

Alors chaque chacune des (n + 1) iterations s'execute en temps
constant. Dans ce cas l'algorithme s'execute en O(n).

Dans le cas il existe au moins un entier k < n tel qu'il n'y
a pas de monome de degre k, alors il suffit de considerer dans
le calcul que le polynome contient un monome 0.X^k, ce qui ne change
rien a la complexite qui a ete calculee.

*)

(***************************************************************)

let rec derivePolynome = function
[] -> []
  | (coeff, 0)::[] -> []
  | (coeff, exp)::tail -> (coeff *. (float_of_int exp), exp - 1)::(derivePolynome tail);;

(***************************************************************)

let rec primitivePolynome = function
[] -> []
  | (coeff, exp)::tail -> (coeff /. (float_of_int exp +. 1.), exp + 1)::(primitivePolynome tail);;

(***************************************************************)

let rec integrePolynome p a b =
  let q = primitivePolynome p in
  evalHorner q b -. (evalHorner q a);;

(***************************************************************)

let rectangles p a b n =
  let nFloat = float_of_int n in
  let rec aux p a b m =
```

```
    (b -. a) *. (evalHorner p a) /. nFloat +.
      ( if (m = 1) then
        0.
      else
        aux p ((b -. a)/. nFloat +. a) b (m-1)
      ) in
  aux p a b n;;

(*****************************************************************)

let trapezes p a b n =
  let nFloat = float_of_int n in
  let dx = (b -. a) /. (float_of_int n) in
  let somme = ref (evalHorner p a +. evalHorner p b) in
  for i = 1 to n-1 do
    let j = float_of_int i in
    somme := 2.*.(evalHorner p (dx*.j))+. !somme;
  done;
  (b -. a) /. (2. *. nFloat) *. !somme;;

(*****************************************************************)

let facteurLagrange x = [(1., 1); (-.x, 0)];;

(*****************************************************************)

let rec lagrange_f_i pts x_i = match pts with
  [] -> polynomeUnite
| (x, y)::t -> if x = x_i then
    lagrange_f_i t x_i
else
    multPolynomes (facteurLagrange x) (lagrange_f_i t x_i);;

(*****************************************************************)

(*
p_i = y_i * f_i(x)/f_i(x_i)
*)

(*****************************************************************)

let lagrange_p_i pts (x_i, y_i) =
  let f_i = lagrange_f_i pts x_i in
  multConst f_i (y_i /. (evalHorner f_i x_i));;

(*****************************************************************)

(*
p(x) = somme des p_i(x)
*)

(*****************************************************************)

let interpolationLagrange pts =
  let rec add_p_i l =
    (match l with
      [] -> polynomeNul
    | h::t -> addPolynomes (lagrange_p_i pts h) (add_p_i t)
    ) in add_p_i pts;;

(*****************************************************************)

let interpolationVandermonde pts =
  let n = List.length pts in
  let x = make_vect n 0. in
  let y = make_matrix n 1 in
  let rec initTab l m =
     match l with
     | [] -> ()
     | (xi, yi)::tail ->
         (
          x.(m) <- xi;
          y.(m).(0) <- yi;
          initTab tail (m+1)
         ) in
  initTab pts 0;
  let a = make_matrix n n in
  fill_matrix a (function (i, j) -> if (j = 0) then 1. else a.(i).(j-1) *. x.(i));
  let x = solve_system a y in
  let rec coeffs k =
    if (k >= 0) then
```

```
      (x.(k).(0), k)::(coeffs (k-1))
    else
      [] in
  coeffs (n-1);;

(*****************************************************************)

(*

u_(n+1) = u_n - f(x)/f'(x)

*)

(*****************************************************************)

let rec newton p x = function
    0 -> x
  | n -> newton p (x -. (evalHorner p x)/.(evalHorner (derivePolynome p) x)) (n - 1);;

(*****************************************************************)

let degre = function
  | [] -> -1
  | (c, e)::_ -> if (e <> 0) then
      e
  else
      if (estNul c) then
        -1
      else
        0;;

(*****************************************************************)

let rec divisionPolynomes a b =
     if (degre a < degre b) then
       (polynomeNul, a)
     else
       (
       match a, b with
         (ca, ea)::_, (cb, eb)::_ ->
        let q = [(ca/.cb, ea-eb)] in
        let r = subPolynomes a (multPolynomes b q) in
        let (qrec, rrec) = divisionPolynomes r b in
        ((addPolynomes q qrec), rrec)
       );;

(*****************************************************************)

let rec pgcd a b =
     if (degre b <= 0) then
       a
     else
        let (_, r) = divisionPolynomes a b in
        pgcd b r;;

let r = multPolynomes (facteurLagrange (-.1.)) (facteurLagrange 4.);;
strOfPolynome r;;
let p = multPolynomes (facteurLagrange 5.) (multPolynomes r (facteurLagrange 2.));;
let q = multPolynomes (facteurLagrange 6.) (multPolynomes r (facteurLagrange 1.));;

strOfPolynome (pgcd p q);;

(*****************************************************************)

let rec euclideE a b lastV =
     if (degre b <= 0) then
       (a, polynomeNul, lastV)
     else
        let (q, r) = divisionPolynomes a b in
        let (p, u, v)= euclideE b r lastV in
        (d, v, subPolynomes u (multPolynomes v q));;

strOfPolynome (euclideE p q polynomeNul);;

(*****************************************************************)

let sturmNext n npUn =
  let q, r = divisionPolynomes n npUn in
  multConst r (-1.);;

let sturmSuite p =
```

```
  let rec sturmSuiteAux u0 u1 =
    let u2 = sturmNext u0 u1 in
    u2::(
        if (degre u2 <= 0) then
          []
        else
          sturmSuiteAux u1 u2
       ) in
  let dp = multConst (derivePolynome p) (-1.) in
  p::dp::(sturmSuiteAux p dp);;

(*************************************************************)

let rec nbSignChange s x = match s with
   [] -> 0
| h::[] -> 0
| h1::h2::t ->
    (
     let h1x = evalHorner h1 x in
     let h2x = evalHorner h2 x in
     if (h1x < 0. && h2x > 0.) or (h1x > 0. && h2x < 0.) then
       1
     else
       0
    ) + (nbSignChange (h2::t) x);;

(*************************************************************)

let rec map f = function [] -> [] | h::t -> (f h)::(map f t);;

(*************************************************************)

let rec trouveRacines s inf sup eps =
  let sinf = nbSignChange s inf in
  let ssup = nbSignChange s sup in
  if (sinf = ssup) then
    []
  else
    let mid = (inf +. sup) /. 2. in
    if (absFloat (sup  -. inf) < eps) then
      [mid]
    else
      (trouveRacines s inf mid eps)@(trouveRacines s mid sup eps);;

(*************************************************************)

let majoreRacines p =
    let maxFloat a b =
      if (a < b) then b else a in
    match p with
  (a_n, _)::_ -> let rec maxCoeff l =
    (
     match l with
      | (h, _)::[] -> absFloat h
      | (h, _)::t -> maxFloat (absFloat h) (maxCoeff t)
      | _ -> failwith "oups"
    ) in (maxCoeff p)/.(absFloat a_n) +. 1.
    | _ -> failwith "oups";;

(*************************************************************)

let sturm p eps =
  let s = sturmSuite p in
  let a = majoreRacines p in
  trouveRacines s (-.a) a eps;;

(*************************************************************)

let rec monstronome n = match n with
  0 -> polynomeUnite
| n-> multPolynomes (expPolynome (facteurLagrange (float_of_int n)) (n mod 4 + 1)) (monstronome (n-1));;

let rec gronome = function
  0 -> [(-.1., 0)]
| n-> (float_of_int ((if n mod 2 = 0 then (-n) else n) / 2) , n)::(gronome (n-1));;

for i=1 to 8 do
  let p = monstronome i in
  print_float (integrePolynome p 0. 5.);
  print_string "\n";
  print_float (rectangles p 0. 5. 100000);
```

```
  print_string "\n";
  print_float (trapezes p 0. 5. 100000);
  print_string "\n";
  print_string (strOfPolynome p);
  print_string "\n";
  print_float (majoreRacines p);
  print_string "\n";
  map (function e -> print_float e; print_string ";") (sturm p 1.e-10);
  print_string "\n";
done;;
```