1. Proposer différentes implémentations du type file (liste, vecteurs circulaires, tas etc).

2. *Les files*

La file est une structure de données essentielle et omniprésente en informatique. C'est une file qui, par exemple, gère les caractères tapés au clavier, les impressions sur l'imprimante, les requêtes sur un réseau etc. Il s'agit d'une structure linéaire dynamique telle que l'élément le plus ancien est traité le premier. On dit que l'on a affaire à une structure de données FIFO (*first in first out*). L'insertion d'un nouvel élément se fait à l'extrémité opposée de celle où l'on retire les plus anciens (certains distributeurs de gobelets fonctionnent ainsi).

• Spécifications

La signature du type abstrait est la suivante :

- Deux constructeurs :
  - `File_vide` de type `unit -> file` qui crée une file vide,
  - `ajoute` de type `\'{e}l\'{e}ment` × `file -> file` qui ajoute un objet de type `\'{e}l\'{e}ment` à la fin de la file,
- Un prédicat :
  - `est_vide` de type `file -> bool` qui teste si une file est vide,
- Deux fonctions de sélection :
  - `premier` de type `file -> \'{e}l\'{e}ment` qui retourne l'objet au début de la file,
  - `queue` de type `file -> file` qui renvoie la file privée de son premier élément.

En désignant par `e` un élément et par `f` une file d'objets de même type que `e`, on a la sémantique :

- `est_vide File_vide = vrai`,
- `est_vide (ajoute e f) = faux`,
- `premier File_vide = erreur`,
- `premier (ajoute e f) =`
  si `(est_vide f)` alors `e` sinon `(premier f)`,
- `queue File_vide = erreur`,
- `queue (ajoute e f) =`
  si `(est_vide f)` alors `File_vide` sinon `(ajoute e (queue f))`.

Le type abstrait `file` est inductif (les fonctions `premier` et `queue` sont d'ailleurs définies par induction).

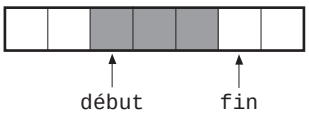
• Implémentations

Le codage des files est un peu plus subtil que celui d'une pile par exemple. En effet, une implémentation naïve par des listes conduit à des fonctions d'ajout et de suppression du premier élément qui ne peuvent opérer toutes les deux en temps constant : l'une des deux doit parcourir complètement la liste. Cette idée est donc abandonnée dans la pratique.

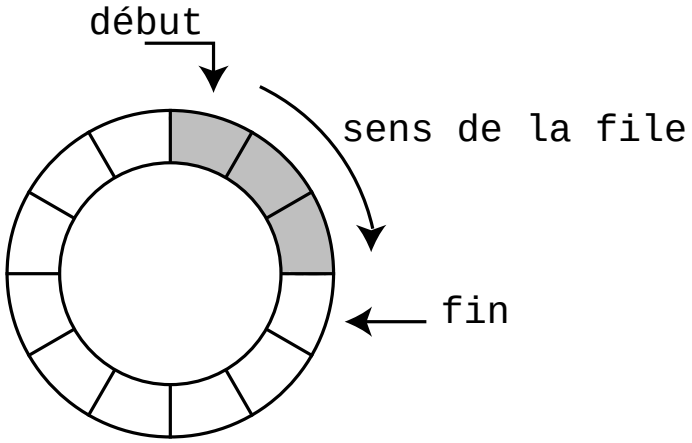
Vous allez implémenter le type `file` à l'aide de vecteurs circulaires. On crée en Caml un type produit :

```
type 'a file = {contenu : 'a vect;
                mutable d\'{e}but : int; mutable fin :int};;
```

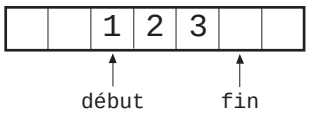
contenant le vecteur proprement dit et deux indices : `d\'{e}but`, qui pointe sur le premier élément de la file, et `fin` qui pointe sur la case *qui suit* le dernier élément. On a `d\'{e}but` $<$ `fin`. Ajouter un élément à la file revient à le placer dans la coordonnée d'indice `fin` et à incrémenter cet entier ; prendre la queue d'une file revient simplement à incrémenter `d\'{e}but`. La figure ci-dessous représente une file de trois éléments avec cette implémentation.



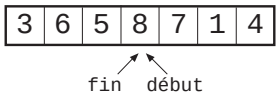
Toutefois un problème se pose : chaque action de `queue` déplace `d\'{e}but` vers la droite du vecteur jusqu'à éventuellement dépasser la taille de celui-ci et conduire à une erreur, alors qu'en fait tout le début du vecteur est libre. On pourrait imaginer régler ce problème par des translations à gauche du contenu de la file à des moments judicieux... on peut également envisager que le vecteur n'est plus linéaire mais circulaire, les éléments étant lus, par exemple, dans le sens des aiguilles d'une montre (voir figure suivante).



Sur le plan mathématique, cela revient à considérer les indices modulo la longueur du vecteur et donc à laisser tomber la contrainte `d\'{e}but` $<$ `fin`. L'indice `d\'{e}but` pointe toujours sur le début de la file et `fin` indique encore la coordonnée en laquelle on peut ajouter un nouvel élément.
La représentation d'une file donnée n'est manifestement pas unique. Par exemple, la file $(1, 2, 3)$ peut être représentée par :



ou par :

Mais alors, que représente la file 

? On ne peut distinguer la file pleine $(8, 7, 1, 4, 3, 6, 5)$ de la file vide $()$. Pour pallier cet inconvénient, nous allons ajouter à notre type Caml un champ booléen `vide`. Nous pouvons à présent donner le détail de notre implémentation du type `file` :

```
type 'a file = {contenu : 'a vect;
                mutable d\'{e}but : int; mutable fin :int;
                mutable vide : bool};;
```

La taille maximale d'une file intervient directement dans le code des fonctions de manipulation. On peut en faire une variable globale :

```
let fmax = 10;;
```

Pour créer une file vide polymorphe, on peut adopter :

```
let File_vide objet =
{contenu = make_vect fmax objet;
 d\'{e}but = 0; fin = 0; vide = true};;
```

ou, si l'on désire se contenter de files d'entiers de longueur bornée par 10 :

```
let File_vide () =
{contenu = make_vect fmax 0;
 d\'{e}but = 0; fin = 0; vide = true};;
```

1° Ecrire les autres fonctions de manipulation. On prendra soin de gérer les deux types d'erreurs : l'accès aux éléments constitutifs d'une file vide et le débordement d'une file pleine.

Par exemple, la création de la file $(1, 2, 3)$ (où 1 est l'élément le plus ancien) doit se faire de la manière suivante :

```
let f = File_vide ();;
ajoute 1 f; ajoute 2 f; ajoute 3;;
```

2° Ajouter au type abstrait la fonction donnant la taille d'une file.

Solution proposée

1°

```
let est_vide f =
    f.vide;;
let ajoute e f =
    if (f.fin = f.d\'{e}but) && not(f.vide)
    then failwith "file pleine"
    else begin
         f.contenu.(f.fin) <- e;
         f.fin <- (succ f.fin) mod fmax;
         f.vide <- false;
         f
         end;;
let premier  f =
    if (f.vide)
    then failwith "file vide"
    else f.contenu.(f.d\'{e}but);;
let queue f =
    if (f.vide)
    then failwith "file vide"
    else begin
         f.d\'{e}but <- (succ f.d\'{e}but) mod fmax;
         if (f.d\'{e}but = f.fin)
         then f.vide <- true;
         f
         end;;
```

2°

```
let taille f =
    if (f.vide)
    then 0
    else let t=(f.fin - f.d\'{e}but) in
         if t>0 then t else t+fmax;;
```

3. On cherche à définir les entiers naturels "à la Peano". Pour cela on définit le type suivant :

```
type entier_naturel = Zero | Succ of entier_naturel;;
```

Ainsi, 2 est codé comme le successeur du successeur de 0 soit `Succ (Succ Zero)`.
Donner les définitions récursives Caml des opérations arithmétiques élémentaires d'addition et de multiplication. À l'exécution, on aura par exemple :

```
#add (Succ (Succ ( Succ Zero))) (Succ (Succ ( Succ Zero)));;
- : entier_naturel = Succ (Succ (Succ (Succ (Succ (Succ Zero)))))

#mul (Succ ( Succ Zero)) (Succ (Succ ( Succ Zero)));;
- : entier_naturel = Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
```

Indications

```
let rec add n m = match m with
    | Zero -> n
    | Succ p -> Succ (add n p);;

let rec mul n m = match m with
    | Zero -> Zero
    | Succ p -> add (mul n p) n;;
```

4. **1°** Définir un type `expression_arithmetique` comportant des constantes entières, les opérateurs binaires d'addition et de multiplication, l'opérateur unaire d'exponentiation et les variables formelles.

```
let expr1 =
     Exponentielle(Addition (Variable "x", Constante 1));;
let expr2 = Addition (Multiplication (Variable "x",
                        Addition (Variable "y", Constante 1)),
                       Constante 2);;
```

L'expression `expr1` (resp. `expr2`) représente $\exp(x+1)$ (resp. $(x*(y+1))+2$). Les flots permettent de coder de façon un peu moins fastidieuse ces expressions.

2° Définir ensuite une dérivation formelle dont voici un exemple d'utilisation :

```
#derive "x" expr1;;
- : expression =
 Multiplication
  (Addition (Constante 1, Constante 0),
   Exponentielle (Addition (Variable "x", Constante 1)))
#derive "x" expr2;;
- : expression =
 Addition
  (Addition
    (Multiplication
          (Constante 1, Addition (Variable "y", Constante 1)),
     Multiplication (Variable "x",
           Addition (Constante 0, Constante 0))),
   Constante 0)
```

On est certes loin encore d'une présentation agréable : la simplification est un problème crucial en calcul formel !

Solution proposée

1°

```
type expression = Constante of int
                | Variable of string
                | Addition of expression * expression
                | Multiplication of expression * expression
                | Exponentielle of expression;;
```

2°

```
let rec derive variable_de_derivation = function
    | (Constante n)  -> (Constante 0)
    | (Variable x)   -> if (x=variable_de_derivation)
                        then (Constante 1)
                        else (Constante 0)
    | (Addition (expr1, expr2)) ->
              Addition (derive variable_de_derivation expr1,
                        derive variable_de_derivation expr2)
    | (Multiplication (expr1, expr2)) ->
       Addition
        (Multiplication (derive variable_de_derivation expr1,
                         expr2),
         Multiplication (expr1,
                         derive variable_de_derivation expr2))
    | (Exponentielle expr) ->
       Multiplication (derive variable_de_derivation expr,
                       Exponentielle (expr));;
derive : string -> expression -> expression = <fun>
```

Une simplification "de base" :

```
let rec simplifie expr = match expr with
    | (Addition (Constante 0, expr1)) -> simplifie expr1
    | (Addition (expr1, Constante 0)) -> simplifie expr1
    | (Addition (Constante n, Constante m)) -> (Constante (m+n))
    | (Multiplication (Constante 0, expr1)) -> (Constante 0)
```

```
    | (Multiplication (expr1, Constante 0)) -> (Constante 0)
    | (Multiplication (Constante 1, expr1)) -> simplifie expr1
    | (Multiplication (expr1, Constante 1)) -> simplifie expr1
    | (Multiplication (Constante n, Constante m)) -> (Constante (m*n))
    | (Exponentielle (Constante 0)) -> (Constante 1)
    | (Constante n)  -> (Constante n)
    | (Variable x)   ->  (Variable x)
    | (Addition (expr1, expr2)) ->  (Addition (simplifie expr1, simplifie expr2))
    | (Multiplication (expr1, expr2)) ->  (Multiplication (simplifie expr1 , simplifie expr2))
    | (Exponentielle expr) -> (Exponentielle (simplifie expr));;
```

ce qui est loin d'être optimal.

5. On se propose d'implémenter efficacement quelques opérations usuelles sur des matrices *creuses*, c'est-à-dire dont les coefficients sont presque tous nuls.
On se fixe une taille arbitraire pour les matrices :

```
let nmax = 4;;
```

On se propose de représenter un matrice comme un vecteur de colonnes et de représenter une colonne par la liste de ses composantes non nulles. Chaque composante comporte un coefficient réel `coeff` et un indice de ligne compris entre 1 et nmax.
Ceci donne en Caml :

```
type mat_creuse == colonne vect
and colonne == composante list
and composante = {mutable coeff : float ; mutable ligne : int};;
```

La création de la matrice nulle est réalisé par la fonction `nulle` :

```
let nulle  () = let mat_nulle = make_vect nmax [] in
for i=0 to (nmax -1) do mat_nulle.(i) <- [] done; (mat_nulle:mat_creuse);;
(* sert a rendre independantes les colonnes *)
```

On décide que les éléments d'une composante sont listées par indice croissant.

1° Écrire une fonction `ajout (m :mat_creuse) x i j` qui ajoute l'élément x à la matrice m en position (i,j).
Voici un exemple d'exécution :

```
#let m0 = nulle ();; ajout m0 1. 1 2; ajout m0 5. 1 4; ajout m0 4. 3 2; ajout m0 8. 2 2 ;;
m0 : mat_creuse = [|[]; []; []; []|]
#- : unit = ()
#m0;;
- : mat_creuse =
 [|[];
   [{coeff = 1.0; ligne = 1}; {coeff = 8.0; ligne = 2};
    {coeff = 4.0; ligne = 3}];
   []; [{coeff = 5.0; ligne = 1}]|]
```

2° Écrire une fonction `transposée` qui retourne la transposée d'une matrice creuse.

3° Écrire une fonction `produit` qui réalise le produit matriciel de deux matrices creuses.

Indications

```
                (*       MATRICE CREUSE : UNE SOLUTION INCOMPLETE         *)


let nmax = 4;;

type mat_creuse == colonne vect
and colonne == composante list
and composante = {mutable coeff : float ; mutable ligne : int};;
```

```
(* attention au bug classique sur l'ind\'{e}pendance des colonnes *)

let nulle  () = let mat_nulle = make_vect nmax [] in
for i=0 to (nmax -1) do mat_nulle.(i) <- [] done; (mat_nulle:mat_creuse);;


let rec insere   c i  = function
| [] -> [{coeff = c ; ligne = i}]
| ({coeff = _ ; ligne = j} :: r as l) when i<=j -> {coeff = c ; ligne = i} :: l
| d :: r  -> d::(insere c i r);;


let ajout (m:mat_creuse) c  i j =  m.(j-1) <- insere c i m.(j-1);;
(* ajout et ajout_colonne procedent par effets de bord *)

let rec ajout_colonne m i  = function
| [] -> ()
| {coeff = c ; ligne = j} :: r -> ajout m c i j ; ajout_colonne m i r;;


let transpos\'{e}e (m:mat_creuse) =
  let tm= nulle () in
  for i=nmax-1 downto 0 do
    ajout_colonne tm  (i+1)  m.(i)
  done;
  tm;;


(* Pour le produit : utiliser une fonction auxiliaire pour le produit scalaire... *)
```

6. Implémenter le type matrice creuse.

Indications

```
                (*       MATRICE CREUSE : UNE SOLUTION INCOMPLETE         *)


let nmax = 4;;

type mat_creuse == colonne vect
and colonne == composante list
and composante = {mutable coeff : float ; mutable ligne : int};;

(* attention au bug classique sur l'ind\'{e}pendance des colonnes *)

let nulle  () = let mat_nulle = make_vect nmax [] in
for i=0 to (nmax -1) do mat_nulle.(i) <- [] done; (mat_nulle:mat_creuse);;


let rec insere   c i  = function
| [] -> [{coeff = c ; ligne = i}]
| ({coeff = _ ; ligne = j} :: r as l) when i<=j -> {coeff = c ; ligne = i} :: l
| d :: r  -> d::(insere c i r);;
```

```
let ajout (m:mat_creuse) c  i j =  m.(j-1) <- insere c i m.(j-1);;
(* ajout et ajout_colonne procedent par effets de bord *)

let rec ajout_colonne m i  = function
| [] -> ()
| {coeff = c ; ligne = j} :: r -> ajout m c i j ; ajout_colonne m i r;;


let transpos\'{e}e (m:mat_creuse) =
  let tm= nulle () in
  for i=nmax-1 downto 0 do
    ajout_colonne tm  (i+1)  m.(i)
  done;
  tm;;


(* un exemple *)
let m0 = nulle ();; ajout m0 1. 1 2; ajout m0 5. 1 4; ajout m0 4. 3 2; ajout m0 8. 2 2 ;;
transpos\'{e}e m0;;

(* Pour le produit : utiliser une fonction auxiliaire pour le produit scalaire... *)


type mat2 == objet list
and objet = {mutable coeff : float ; mutable ligne : int ; mutable col : int ;
             mutable voisin_meme_col : int ; mutable voisin_meme_ligne : int};;

let nmax=6;;

let ajout2 m c i j l k =  {coeff = c ; ligne = i; col = j ; voisin_meme_col = l ;
                           voisin_meme_ligne = k} :: m;;

(* on part de bas en haut et de gauche \`{a} droite *)
let conversion m =
  let mc = ref [] in
  let ligne = make_vect nmax 0 in
  let colonne = make_vect nmax 0 in
      for i=(nmax-1) downto 0 do
      for j=(nmax-1) downto 0 do
        if m.(i).(j) <> 0.0 then begin
          mc := ajout2 !mc m.(i).(j)  (i+1) (j+1) (colonne.(j)) (ligne.(i));
          ligne.(i) <- j+1; colonne.(j) <- i+1
      end done  done;
(!mc:mat2);;


(* un exemple *)
let m1 = [| [|0.;0.;7.;0.;0.;4.|] ; [|0.;0.;0.;0.;0.;0.|] ; [|0.;0.;0.;0.;0.;0.|] ;
[|0.; 0.; 5.; 0.; 0.; 0.|] ; [|0.;0.;0.;0.;0.;0.|] ;[|0.;0.;0.;0.;0.;0.|] |];;
conversion m1;;
```

7. *Tas de sable (ENS Lyon Math-info 99)*
On appelle *configuration* une matrice M de dimensions $p \times q$ à coefficients entiers naturels. Si un élément $m_{i,j}$ est $\geqslant 4$, on peut lui retrancher 4 et ajouter 1 à chacun de ses voisins immédiats (les éléments $m_{i-1,j}$ si $i \neq 1$, $m_{i+1,j}$ si $i \neq p$,

$m_{i,j-1}$ si $j \neq 1$ et $m_{i,j+1}$ si $j \neq q$). Si M' est le résultat de cette transformation, on écrit $M \to M'$ ou, plus précisément, $M \xrightarrow{i,j} M'$.

1° Montrer que la relation $\to$ termine ; autrement dit, quelle que soit la configuration de départ et la suite de transformations effectuées, on arrive à une configuration bloquée dans laquelle aucun élément n'est $\geqslant 4$.

2° Peut-on avoir deux configurations bloquées distinctes à partir d'une même configuration de départ M ?

3° Mêmes questions avec des configurations sans bord (matrices ■ infinies ■).

4° Proposer une implémentation CAML

Solution proposée

1° Pour une configuration donnée, on note

$$h(M) = \sum_{i,j} m_{i,j} \text{ et } w(M) = \sum_{i,j} (i+j-2)^2 m_{i,j}.$$

(on pourrait ne pas écrire le -2, la majoration à venir serait alors moins bonne...)
Pour une transformation $M \to M'$ sur un site (i,j) non périphérique ($1 < i < p$, $1 < j < q$), on a

$$h(M') = h(M) \text{ et } w(M') - w(M) = 2(i+j-3)^2 + 2(i+j-1)^2 - 4(i+j-2)^2 = 4.$$

Donc, si $M^{(0)} \to \cdots \to M^{(n)}$ est une suite de transformations non périphériques, alors

$$4n \leqslant w(M^{(n)}) = \sum_{i,j} (i+j-2)^2 m_{i,j}^{(n)} \leqslant (p+q-2)^2 h(M^{(n)}) = (p+q-2)^2 h(M^{(0)}).$$

Partant de M une suite de $\left\lfloor \frac{1}{4}(p+q-2)^2 h(M) \right\rfloor + 1$ transformations consécutives contient au moins une transformation périphérique qui diminue d'au moins 1 la quantité $h(M)$. Au total, une telle suite issue de M contient donc au plus $h(M)\left(\left\lfloor \frac{1}{4}(p+q-2)^2 h(M) \right\rfloor + 1\right)$ transformations.

2° Non. Comme $\to$ termine, on peut démontrer ce résultat en supposant qu'il est vrai pour toutes les configurations M' telles que $M \to M'$ (c'est une preuve par induction : cf l'exercice sur la confluence).
Soient $M \xrightarrow{*} M_1$ et $M \xrightarrow{*} M_2$ deux suite de transformations avec M_1 et M_2 bloquées. Si M n'est pas bloquée, il existe un couple (i,j) tel que $m_{i,j} \geqslant 4$. Pour $k = 1, 2$, l'une au moins des transformations $M \xrightarrow{*} M_k$ est la transformation (i,j). Or, comme une transformation équivaut à ajouter à la matrice une autre matrice (avec un -4 et au plus quatre 1), les transformations commutent et on a donc $M \xrightarrow{i,j} M' \xrightarrow{*} M_k$. En appliquant l'hypothèse à M', on obtient $M_1 = M_2$.

3° Les résultats précédents sont encore valables.
Montrons d'abord la terminaison dans le cas où la configuration initiale est un carré d'éléments tous égaux. Plus précisément, $m_{i,j} = t$ pour $|i| \leqslant p$ et $|j| \leqslant p$ et $m_{i,j} = 0$ pour les autres $(i,j) \in \mathbb{Z} \times \mathbb{Z}$; où $p, t \in \mathbb{N}^*$. Soit $M = M^{(0)} \to M^{(1)} \to \cdots$ est une suite infinie de transformations. Pour tout entier N, d'après la question 1, il existe un rang K pour lequel les éléments $\geqslant 4$ de $M^{(K)}$ sont en dehors du carré $|i|, |j| \leqslant N$. Mais on voit par récurrence que, pour tout k, d'une part il existe au moins un élément non nul de $M^{(k)}$ dans chacun des quatre quadrants du plan, et d'autre part les éléments non nuls de $M^{(k)}$ forment un ensemble connexe (en un sens évident). Ainsi, si $m_{i_0,j_0}^{(K)} \geqslant 4$ et si $m_{i_1,j_1}^{(K)} \neq 0$ où (i_1, j_1) se trouve dans le quadrant oppposé à celui de (i_0, j_0), il existe un chemin de (i_0, j_0) à (i_1, j_1) formé d'éléments non nuls de $M^{(K)}$ et, par conséquent, $h(M) = h(M^{(K)}) \geqslant N$: il suffit donc de choisir $N > h(M)$ pour obtenir une contradiction.
Dans le cas d'une configuration M quelconque, il suffit de considérer une configuration du type précédent $\geqslant$ à M.
Le fait qu'une configuration n'a qu'un seul réduit se déduit alors du cas borné.

8. Déterminer et prouver le résultat calculé par la fonction suivante :

```
let f x y =
    let r = ref 0 and s = ref y in
        while (!s >0) do
          r:= !r + x;
          s:= !s -1
        done;
    s:= y -1;
    let t = ref !r in
        while (!s>0) do
          r:= !r + !t;
          s:= !s -1
        done;
    !r;;
```

Solution proposée

Il y a deux boucles à considérer. Définissons (avec les conventions usuelles de notation) comme premier invariant de boucle :

$$r_i + xs_i = xy$$

On a bien $r_0 = 0$ et $s_0 = y$ soit l'invariant pour $i = 0$. Ensuite si l'on suppose la relation après i itérations de la première boucle, comme $r_{i+1} = r_i + x$ et $s_{i+1} = s_i - 1$, on a $r_{i+1} + xs_{i+1} = r_i + x + x(s_i - 1) = r_i + xs_i = xy$. □

À la fin de la première boucle, `!s = 0` donc `!t = !r = x y`.

On définit alors un deuxième invariant de boucle :

$$r_i + xys_i = xy^2$$

Il est à nouveau vérifié à chaque étape car avant la seconde boucle $r_0 = xy$ et $s_0 = y-1$ puis, en supposant la relation après i itérations de boucle, comme $r_{i+1} = r_i + !t = r_i + xy$ et $s_{i+1} = s_i - 1$, on a $r_{i+1} + xys_{i+1} = r_i + xy + xy(s_i - 1) = r_i + xys_i = xy^2$. □

Finalement, comme la seconde boucle termine lorsque `!s=0`, la valeur retournée par `f` est `!r` $= xy^2$.

9. Déterminer et prouver le résultat calculé par la fonction suivante :

```
let t a =
    let x = ref a and y = ref a in
    while not(!y=0) do
      x:= !x + 2;
      y:= !y -1
    done;
!x;;
```

Indications

L'invariant à considérer est $x_i + 2y_i = 3a$ et la fonction calcule le triple de a.

10. **1°** Que fait le programme suivant ?

```
let p a n =
    let r= ref 1 and b = ref n and c = ref a in
    while (!b >0) do
          if (!b mod 2 = 1)
          then r := !r * !c;
          b := !b / 2;
          c := !c * !c
    done;
    !r;;
```

Prouvez votre affirmation.

2° Donnez une version récursive du même programme.

Indications

1° Il s'agit de l'exponentiation rapide. La référence `c` (pour carré) représente successivement a, a^2, a^4... et les chiffres de l'écriture binaire de n sont testés avec `b` (pour base2). Considérez l'invariant de boucle $r_i \cdot (carré_i)^{b_i} = a^n$. pour prouver ce programme.

2° Le programme récursif se comprend sans commentaire :

```
let rec puissance_rapide a  = function
    | 0 -> 1
    | n -> let r= puissance_rapide a (n / 2) in
                  if (n mod 2)=0 then r*r else a*r*r;;
```

et la complexité est inchangée.

11. Soit

```
let rec v n  = if n=0 then 0 else n-v(v(n-1));;
```

Montrer que le calcul de `v` termine. Que se passe-t-il lorsque $v(0) = 1$?

Indications

Montrer que pour tout $k \in [\![0, n]\!]$, on peut calculer $v(k)$ et $v(k) \in [\![0, k]\!]$...

12. Sous séquence commune.

On cherche à déterminer la longueur d'une sous suite commune de longueur maximale entre deux mots $u = a_1 a_2 \dots a_n$ et $v = b_1 b_2 \dots b_m$ (attention : les lettres communes ne sont pas forcément consécutives).
Pour $1 \leqslant j \leqslant m$ et $0 \leqslant i \leqslant n$, on note $L(i, j)$ la longueur d'une sous séquence commune maximale entre $a_1 \dots a_i$ et $b_1 \dots b_j$.

1° Montrer que :

$$L(i,j) = \begin{cases} 1 + L(i-1, j-1) \\ max(L(i, j-1), L(i-1, j))) \end{cases}$$

2° En déduire une fonction Caml de calcul de L (on implémentera des vecteurs d'entiers).

Solution proposée

1° Soit w une sous séquence de longeur maximale commune à $a_1 \dots a_{i-1}$ et à $b_1 \dots b_{j-1}$:
– si $a_i = b_j$ wa_i est une sous séquence commune maximale à $a_1 \dots a_i$ et $b_1 \dots b_j$,
– si $a_i \neq b_j$ alors une sous séquence commune maximale à $a_1 \dots a_i$ et $b_1 \dots b_j$ est ou bien commune à $a_1 \dots a_i$ et $b_1 \dots b_{j-1}$ (si elle ne se termine pas par b_j) ou bien à $a_1 \dots a_{i-1}$ et $b_1 \dots b_j$ (si elle ne se termine pas par a_i)
ce qui explique la relation annoncée.

2° Voici une fonction récursive Caml dont la terminaison et la correction sont immédiates :

```
let u = [|1; 8; 3; 4; 5; 6; 7; 8; 9|];;

let v = [|1; 3; 7; 4; 5; 8; 9; 7|];;

let rec long = fun
| i  0  -> 0
| 0  j  -> 0
| i  j  when u.(i-1) = v.(j-1) -> 1 + long (i-1)  (j-1)
| i  j  -> max (long i (j-1)) (long (i-1) j);;

#long 3 3;;
- : int = 2
#long 9 8;;
- : int = 6
```

Évidemment, il faut que l'appel de long se fasse avec des valeurs cohérentes de i et j. On peut pallier cet inconvénient avec des listes (lues de droite à gauche) :

```
let rec longbis u v = match (u,v) with
| (_,[]) -> 0
| ([],_) -> 0
| (a::r, b::s) when a=b -> 1+longbis r s
| (a::r, b::s)  -> max (longbis (a::r) s) (longbis r (b::s));;

#longbis [3; 8; 1] [7; 3; 1];;
- : int = 2
#longbis [9; 8; 7; 6; 5; 4; 3; 8; 1] [7; 9; 8; 5; 4; 7; 3; 1];;
- : int = 6
```

13. Correction et terminaison du problème des mariages stables

14. La fonction de McCarthy est définie par :

```
let rec f n =
    if n>100
    then n-10
    else f (f (n+11));;
```

La fonction `f` termine-t-elle sur $\mathbb{Z}$? Si tel est le cas que vaut-elle ?

Solution proposée

Fonction de McCarthy.

Nous allons montrer que le calcul de `f (x)` termine toujours sur $\mathbb{Z}$ et vaut : $x-10$ si $x>100$ et 91 si $x \leqslant 100$.

L'intervalle d'entiers $]\!]100, +\infty[\![$ va constituer notre ensemble $\mathcal{B}$ de cas de base ; nous allons en effet, pour $x \leqslant 100$, procéder par induction sur $(]\!]-\infty, 100]\!], \geqslant)$ qui est bien fondé (car majoré).

Nous allons définir le prédicat suivant :

$\mathfrak{p}_f(x)$ = "$f(x)$ termine et

- si $100 < x$, $f(x) = x - 10$,
- si $x \leqslant 100$, $f(x) = 91$"

et le montrer en suivant le théorème de correction (attention, l'ordre considéré est l'opposé de l'ordre naturel sur les entiers ; notre preuve inductive va donc aller *a contrario* de nos habitudes de preuve par récurrence). Trois cas se présentent :

- soit $x > 100$, $f(x)$ termine et vaut bien $x-10$, c'est un cas de base.
- soit $90 \leqslant x \leqslant 100$ et par définition $f(x) = f(f(x+11))$. Comme $101 \leqslant x+11$, $f(x+11) = x+11-10 = x+1$ et donc dans cet intervalle $f(x) = f(x+1)$. On peut recommencer ce raisonnement sur $x+1$, puis $x+2$... tant que l'on reste dans l'intervalle $[\![90, 100]\!]$. On obtient alors $f(x) = f(x+1) = ... = f(100) = f(101)$. Car $f(101)$ constitue le cas d'arrêt rencontré. La valeur calculée vaut donc $f(101) = 91$. Le prédicat est donc à nouveau vérifié dans ce cas.
- soit $x < 90$ et $f(x)$ conduit encore au calcul de $f(f(x+11))$. Comme $101 \geqslant x+11 > x$, par hypothèse d'induction $f(x+11)$ termine et vaut $91 > x$. Donc $f(x)$ termine et vaut $f(f(x+11)) = f(91) = 91$. Le prédicat est donc à nouveau vérifié dans ce cas.

Finalement, on a bien montré, en suivant le théorème de correction, le résultat annoncé.

15. *Mariage stable*

On cherche à marier n hommes à n femmes. Un mariage est donc une bijection de l'ensemble des hommes dans l'ensemble des femmes. Chaque homme a une liste de préférences qui ordonne totalement l'ensemble des femmes, et réciproquement. On dit qu'un mariage est stable s'il n'existe pas une femme f et un homme h' tels que f préfère h' à son conjoint, et réciproquement h' préfère f à sa conjointe (autrement dit, on veut éviter tout risque d'adultère).

1° On considère l'algorithme de mariage suivant.

```
R\'{e}p\'{e}ter
    Chaque homme demande en mariage la femme la plus haute sur sa
       liste qui ne l'a pas d\'{e}j\`{a} rejet\'{e}.
    Toute femme qui re\c{c}oit k>1 demandes en mariage rejette les
       demandes \'{e}manant des k-1 hommes les plus bas sur sa liste.
Jusqu'\`{a} ce que chaque femme re\c{c}oive exactement une demande.
On marie alors chaque femme \`{a} l'homme qui l'a demand\'{e}e.
```

Quel est le mariage produit par l'algorithme sur l'exemple suivant ?

$Hommes : \{h_1, h_2, h_3, h_4\}$ $Femmes : \{f_1, f_2, f_3, f_4\}$

- $h_1 : f_1 > f_2 > f_3 > f_4$ $f_1 : h_3 > h_1 > h_2 > h_4$
- $h_2 : f_1 > f_3 > f_2 > f_4$ $f_2 : h_2 > h_4 > h_1 > h_3$
- $h_3 : f_3 > f_4 > f_1 > f_2$ $f_3 : h_4 > h_1 > h_2 > h_3$
- $h_4 : f_3 > f_2 > f_1 > f_4$ $f_4 : h_1 > h_2 > h_3 > h_4$

2° Montrez que cet algorithme termine toujours et produit un mariage stable.

3° Montrez que cet algorithme produit toujours un mariage M qui est optimal du point de vue des hommes. C'est-à-dire que si M' est un autre mariage stable, aucun homme n'est marié dans M' à une femme qu'il préfère à son épouse dans M. Réciproquement, montrez que ce mariage stable est le plus mauvais du point de vue des femmes.

Solution proposée

1°

- itération 1 : f_1 rejette h_2, f_3 rejette h_3.
- itération 2 : f_3 rejette h_2.
- h_1 épouse f_1, h_2 épouse f_2, h_3 épouse h_4, h_4 épouse f_3.

2° On remarque qu'un homme qui a demandé une femme en mariage continue à demander la même femme à toutes les itérations ultérieures jusqu'à être rejeté, ou marié à la fin de l'algorithme. Toute femme demandée en mariage à une certaine itération est donc demandée également à toutes les itérations ultérieures (éventuellement par des hommes différents). Ceci permet de montrer qu'à chaque étape tous les hommes font une demande (a priori on pourrait imaginer qu'un homme soit rejeté par toutes les femmes, et il ne pourrait plus faire de demandes aux itérations ultérieures).
En effet, supposons par l'absurde qu'un homme h est rejeté par toutes les femmes. Aux itérations ultérieures toutes les femmes reçoivent une demande puisqu'elles ont un jour été demandées par h. C'est impossible sinon il y aurait plus d'hommes que de femmes.
Terminaison : au départ la somme des longueurs des listes de chaque homme est n^2. Si à la fin d'une itération la condition d'arrêt n'est pas vérifiée c'est qu'au moins une femme est demandée par plusieurs hommes (car comme on l'a vu, à chaque étape tous les hommes font une demande). Dans ce cas au moins un homme est rejeté, et la somme des longueurs des listes des femmes demandables diminue d'autant. Il ne peut donc y avoir plus de n^2 itérations. L'algorithme termine donc toujours, et produit un mariage.
Stabilité : Supposons que f et h' sont tels que h' préfère f à sa conjointe f'. C'est donc que f a rejeté h'. Et comme f préfère son conjoint à tous ceux qu'elle a rejetés, alors f préfère en particulier son conjoint à h', et il n'y a donc pas de danger d'adultère.

3° Supposons que M n'est pas optimal. C'est qu'il existe un autre mariage stable M' et un couple (h, f') marié dans M' tel que f' a rejeté h dans l'algorithme. Si f' a rejeté h, c'est qu'elle a été demandée en même temps par un homme h' qu'elle préfère à h. Soit f" la femme de de h' dans M'. Par stabilité de M', h' préfère son épouse f" à f'. A partir du couple (h, f') nous avons donc construit un couple (h', f") vérifiant les mêmes propriétés : h' est marié à f" dans M' mais f" a rejeté h' dans l'algorithme (puisque dans M , h' se retrouve au mieux marié à f'). De plus, le rejet de h' par f" a eu lieu à une itération antérieure au rejet de h par f'. En effet h' ne demande f' en mariage qu'après avoir été rejeté par f". On arrive donc à une contradiction en choisissant le couple (h, f') de sorte que le rejet de h par f' ait lieu le plus tôt possible pendant le déroulement de l'algorithme.
La propriété de pessimalité du point de vue des femmes découle de celle d'optimalité du point de vue des hommes. Supposons en effet que f est mariée à h dans M , et est mariée dans un autre mariage stable M' a un homme h' qu'elle aime moins que h. Dans M', h est marié à une femme f' qu'il préfère à f par stabilité de M'. Mais M ne serait alors pas optimal puisque h y est marié à f. Ouf !
Naturellement, on peut obtenir un mariage stable optimal pour les femmes, et pessimal pour les hommes, avec l'algorithme inversé dans lequel ce sont les femmes qui font les propositions : il n'y a pas que des machos dans ce bas monde...

16. *Fractions égyptiennes (ENS Lyon, Maths-Info)*
Les égyptiens, pour représenter une fraction, n'utilisaient que des sommes d'inverses d'entiers tous distincts. Par exemple, $\frac{5}{6} = \frac{1}{2} + \frac{1}{3}$.

1° Montrer que, pour $r \in]0,1[\cap\mathbb{Q}$, une telle écriture est toujours possible. On proposera un algorithme décomposant $\frac{a}{b}$ avec $0 < a < b$ que l'on testera sur $\frac{10}{13}$. Cette décomposition est-elle unique ? (prouver la terminaison de votre algorithme).

2° Cas où $r \geqslant 1$.

Solution proposée

1° On propose trois méthodes de calcul d'un développement égyptien de $r \in]0,1[\cap\mathbb{Q}$.

a. La première méthode consiste à faire évoluer un *développement partiel* de r, c'est-à-dire une suite croissante $D = (x_1, \ldots, x_m)$ d'entiers telle que

$$r = \frac{1}{x_1} + \cdots + \frac{1}{x_m}.$$

Au départ, si $r = \frac{a}{b}$ où $0 < a < b$, on pose

$$D = (\underbrace{b, \ldots, b}_{a}).$$

Le développement $D = (x_1, \ldots, x_m)$ est égyptien si ses éléments sont tous distincts. Tant que ce n'est pas le cas, on choisit k tel que $x_k = x_{k+1}$ et
– si x_k est pair, on remplace x_k par $\frac{x_k}{2}$ et on supprime x_{k+1} ;
– si $y = x_k$ est impair, on utilise l'identité

$$\frac{2}{y} = \frac{1}{(y+1)/2} + \frac{1}{y(y+1)/2},$$

autrement dit, on supprime de D x_k et x_{k+1} et on les remplace par $(y+1)/2$ et $y(y+1)/2$.
Si on munit l'ensemble des suites finies d'entiers naturels de l'ordre lexicographique, poids fort en tête, le développement D diminue strictement à chaque étape. Cela prouve l'arrêt de l'algorithme.
Avec cette méthode, on obtient

$$\frac{10}{13} = \frac{1}{2} + \frac{1}{7} + \frac{1}{14} + \frac{1}{23} + \frac{1}{91} + \frac{1}{2093}.$$

b. La deuxième méthode de développement de $r = \frac{a}{b}$ est un algorithme glouton dû à Fibonacci.

Si $r = 0$, c'est fini. Sinon, on pose $p = \left\lceil \frac{1}{r} \right\rceil$, de sorte que $\frac{1}{p}$ est maximum vérifiant $\frac{1}{p} \leqslant r$. On continue ensuite avec $r' = r - \frac{1}{p}$.

On montre que les dénominateurs p obtenus forment une suite strictement croissante : Si $r' \neq 0$, en notant $p' = \left\lceil \frac{1}{r'} \right\rceil$, il vient $\frac{1}{p'} \leqslant r - \frac{1}{p} < \frac{1}{p-1} - \frac{1}{p}$; d'où $p' > p(p-1) \geqslant p$.

Pour montrer la terminaison, il suffit de remarquer que le numérateur de $r' = \frac{ap-b}{pb}$ est $<$ à celui de $r = \frac{a}{b}$, ce qui se déduit de l'inégalité $\frac{a}{b} < \frac{1}{p-1}$.
On obtient par exemple

$$\frac{10}{13} = \frac{1}{2} + \frac{1}{4} + \frac{1}{52}.$$

c. Pour la troisième méthode, si a et b sont premiers entre eux, il existe $b' \in [\![0, b-1]\!]$ vérifiant $ab' \equiv 1b \equiv$ modulo p uis a' tel que $ab' = 1 + a'b$ (Bezout). Alors $\frac{a}{b} = \frac{1}{bb'} + \frac{a'}{b'}$ et on continue.
Les dénominateurs bb' obtenus forment une suite strictement décroissante car $b'' < b' < b$ donc $b'b'' < bb'$.
Enfin la méthode termine car car $b' < b$.
Par exemple

$$\frac{10}{13} = \frac{1}{52} + \frac{1}{12} + \frac{1}{6} + \frac{1}{2}.$$

2° Montrons que tout $r \in \mathbb{Q}_+^*$ est développable en fractions égyptiennes. Comme la série harmonique diverge, il existe un entier n vérifiant

$$\sum_{k=2}^{n} \frac{1}{k} \leqslant r \text{ et } \sum_{k=2}^{n+1} \frac{1}{k} > r.$$

Ainsi

$$r = \sum_{k=2}^{n} \frac{1}{k} + r' \text{ avec } 0 \leqslant r' < \frac{1}{n+1}.$$

Les dénominateurs d'un développement de r' en fractions égyptiennes sont donc $> n+1$ et on obtient ainsi un développement de r.

17. On s'intéresse aux mots non vides constitués uniquement des lettres B et N. On définit sur ces mots la relation $X\mathcal{R}Y$ si et seulement si le mot Y s'obtient en remplaçant dans le mot X une séquence BN par N, ou NB par N, ou BB par B, ou NN par B

1° Montrer que toute suite telle que $X_n \mathcal{R} X_{n+1}$ termine.

2° Montrer que toutes les suites commençant par un même mot X_0 terminent sur le même mot.
On se propose de généraliser ce résultat. $\to$ est une relation binaire sur un ensemble $\mathcal{X}$; on note $\xrightarrow{*}$ la fermeture réflexive et transitive de $\to$. On donne les définitions suivantes :
$\to$ *termine* si il n'existe pas de suite $(x_n)_{n\geqslant 0}$ telle que $\forall n \in \mathbb{N}$, $x_n \to x_{n+1}$.
$\to$ est *localement confluente* si

$$\forall x, y_1, y_2 \in \mathcal{X},\ (x \to y_1 \text{ et } x \to y_2) \Rightarrow \exists z \in \mathcal{X},\ y_1 \xrightarrow{*} z \text{ et } y_2 \xrightarrow{*} z.$$

$\to$ est *globalement confluente* si

$$\forall x, y_1, y_2 \in \mathcal{X},\ (x \xrightarrow{*} y_1 \text{ et } x \xrightarrow{*} y_2) \Rightarrow \exists z \in \mathcal{X},\ y_1 \xrightarrow{*} z \text{ et } y_2 \xrightarrow{*} z.$$

3° Montrer que si la relation $\to$ termine et est localement confluente, alors elle est globalement confluente.

4° Montrer que la relation $\mathcal{R}$ est globalement confluente.

5° Montrer que, sans la terminaison, la confluence locale n'implique pas la confluence globale.

Solution proposée

1° La longueur du mot X_n décroît strictement.

2° Si $G = \{b, n\}$ est le groupe à deux éléments (b neutre, et n son propre inverse : $nn = b$), soit x l'élément de G obtenu en remplaçant B par b et N par n dans le mot X_0. Si $x = b$ (resp. n), alors toute suite de premier mot X_0 termine sur B (resp. N).

3° Notons $P(x)$ la propriété (confluence globale)

$$\forall y_1, y_2 \in \mathcal{X},\ (x \xrightarrow{*} y_1 \text{ et } x \xrightarrow{*} y_2) \Rightarrow \exists z \in \mathcal{X},\ y_1 \xrightarrow{*} z \text{ et } y_2 \xrightarrow{*} z.$$

On considère la fermeture réflexive et transitive de $\to$. Comme la relation $\to$ termine, elle est antisymétrique (pas de boucle) et définit ainsi un ordre bien fondé ($M \to M' \iff M' \leqslant M$). On peut donc tenter de montrer que

$$\text{terminaison + confluence locale } \implies \text{ confluence globale}$$

par induction.
On va donc supposer que $P(y)$ est vraie pour tous les y tels que $x \xrightarrow{+} y$ et montrer $P(x)$. Soient donc y_1 et y_2 tels que $x \xrightarrow{*} y_1$ et $x \xrightarrow{*} y_2$. Si $y_1 = x$ ou si $y_2 = x$, l'existence de z est immédiate. Sinon, il existe y_1' et y_2' tels que $x \to y_1' \xrightarrow{*} y_1$ et $x \to y_2' \xrightarrow{*} y_2$. Comme $\to$ est localement confluente, il existe z' vérifiant $y_1' \xrightarrow{*} z'$ et $y_2' \xrightarrow{*} z'$. Par hypothèse d'induction, il existe alors z'' tel que $y_1 \xrightarrow{*} z''$ et $z' \xrightarrow{*} z''$; puis z tel que $z'' \xrightarrow{*} z$ et $y_2 \xrightarrow{*} z$. $P(x)$ est bien démontré.

Ce résultat montre que $P(x)$ est vérifiée pour tout x car s'il existe un x_0 ne vérifiant pas $P(x_0)$, on peut trouver un x_1 ne vérifiant pas $P(x_1)$ et tel que $x_0 \xrightarrow{+} x_1$ puis un x_2 ne vérifiant pas $P(x_2)$ et tel que $x_1 \xrightarrow{+} x_2$ et ainsi de suite ; on contredit ainsi la terminaison.

4° Conséquence de la question 2.

5° Contre-exemple : voici une relation localement confluente mais non globalement confluente. On représente le graphe de la relation par ses aretes : $\mathcal{X} = \{(a,b),(b,a),(a,c),(b,d)\}$. a a deux formes terminales d et c la relation n'est donc pas globalement confluente pourtant en ne considérant que les réécritures en une étape a se réécrit en b ou c et de b on peut rejoindre c en repassant par a. Idem en condiérant b. La relation est donc bien localement conluente. Bien sûr, il n'y a pas terminaison ici à cause de la boucle $(a,b),(b,a)$.

18. Mots de Dyck

Soit $\mathcal{D}$ l'ensemble des mots sur l'alphabet $\mathcal{A} = \{a,b\}$ défini inductivement par :
- $\mathcal{B} = \{\varepsilon\}$,
- $\mathcal{C} = \{\varphi\}$ où φ est d'arité 2 et $\varphi(x,y) = xayb$.

1° Montrer que tout mot de $\mathcal{D}$ est de longueur paire.

2° Donner l'exemple de mots de longueur paire mais n'appartenant pas à $\mathcal{D}$.

3° Montrer que

$$x \in \mathcal{D} \iff (|x|_a = |x|_b \text{ et } |y|_a \geqslant |y|_b \text{ pour tout facteur gauche } y \text{ de } x)$$

4° Considérez une formule logique parenthésée construite à l'aide des connecteurs usuels et de variables propositionnelles, effacez les connecteurs et les variables propositionnelles : qu'obtenez vous ?

Indications

1° immédiat par induction structurelle.

2° aa !

3° $\Rightarrow$: $|x|_a = |x|_b$ se montre immédiatement par induction structurelle. Le résultat sur les prefixes y se montre en considérant tous les cas possibles pour un préfixe d'un mot $uavb$ le résultat étant acquis sur les mots de Dyck u et v par hypothèse d'induction.
$\Leftarrow$, on interprète un mot x vérifiant la propriété $\mathcal{P}$ "$(|x|_a = |x|_b$ et $|y|_a \geqslant |y|_b$ pour tout facteur gauche y de $x)$" comme une "montagne russe" une lettre a correspondant à un un pas "en haut à droite" et une lettre b correspondant à un pas "en bas à droite". La propriété signifie alors que la montagne russe démarre à l'altitude 0 et termine à l'altitude 0 en restant positive (il est facile de vérifier d'ailleurs qu'un tel mot x commence par a et termine par b).
On fait alors une récurrence sur la longueur du mot x considéré : pour conclure il suffit de décomposer x en $uavb$. C'est l'interprétation à l'aide des montagnes russes qui fournit la décomposition : v correspond au mot de longueur maximum qui reste à une altitude $\geqslant 1$, le a qui le précède correspond à la première descente au niveau 0 en lisant x depuis la droite.

4° le mot obtenu appartient au langage de Dyck sur $\mathcal{A} = \{(,)\}$.

19. Nombres de Catalan.
On s'intéresse au langage L défini inductivement sur l'alphabet $A = \{a, b\}$ par les deux règles suivantes :
– $a \in L$
– si x et y sont dans L, xyb est aussi dans L

1° Vérifier que tout mot de L est de longueur impaire. On note C_n le nombre de mots de L de longueur $2n+1$.

2° Que valent les premières valeurs des C_i ? Montrer que

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}$$

3° Montrer que $C_n \leqslant 2^{2n+1}$. En déduire que la SE $S(x) = \sum_{n\geqslant 0} C_n x^n$ a un rayon de convergence > 0. Montrer que cette série vérifie :

$$xS(x)^2 = S(x) - 1$$

et en déduire que $C_n = \frac{1}{n+1}\binom{2n}{n}$.

4° $(C_n)_{n\in\mathbb{N}}$ est la suite des *nombres de Catalan*. Quelles interprétations pouvez vous donner de ces nombres ?

Indications

1° Une énumération facile nous donne $C_0 = C_1 = 1$, $C_2 = 2$ et $C_3 = 5$, $C_4 = 14$. On établit sans peine $C_{n+1} = \sum_{0\leqslant k\leqslant n} C_k C_{n-k}$ ce qui montre que $(C_n)_{n\in\mathbb{N}}$ est définie par une récurrence complète. On peut établir la formule $C_n = \frac{1}{n+1}\binom{2n}{n}$, la preuve la plus simple repose sur l'examen du $DL_n(0)$ de $x \mapsto \sqrt{1-4x}$ si on ne connait pas les séries entières...

2° $S(x) = \dfrac{1-\sqrt{1-4x}}{2x}$.

3° expression postfixée, arbres binaires ...

20. Montrer que la longueur d'une proposition F qui comporte $b(F)$ occurrences de symboles de connecteurs binaires et $n(F)$ occurrences du symbole de négation est $|F| = 4b(F) + n(F) + 1$.

Solution proposée

Longueur d'une proposition en fonction du nombre de connecteurs.
Rappelons que $b(F)$ et $n(F)$ désignent respectivement le nombre d'occurrences de symboles de connecteurs binaires et le nombre d'occurrences du symbole de négation dans l'expression de F. On va montrer par induction que :

$$|F| = 4b(F) + n(F) + 1 \qquad (R)$$

– Si $F \in \mathcal{V}$ c'est-à-dire si F est une variable propositionnelle, alors $b(F) = n(F) = 0$ et $|F| = 1$ et la relation (R) est vérifiée.
– Si $F = \neg G$ avec $|G| = 4b(G) + n(G) + 1$ (hypothèse d'induction) alors $b(F) = b(G)$, $n(F) = n(G) + 1$, $|F| = |G| + 1$ et en appliquant l'hypothèse d'induction on trouve $|F| = |G| + 1 = 4b(G) + n(G) + 1 + 1 = 4b(F) + n(F) + 1$. La relation (R) est à nouveau vérifiée.
– Si $F = (G \top H)$ où $\top$ désigne un connecteur binaire quelconque, avec $|G| = 4b(G) + n(G) + 1$ et $|H| = 4b(H) + n(H) + 1$ (hypothèse d'induction), alors $b(F) = b(G) + b(H) + 1$, $n(F) = n(G) + n(H)$ et $|F| = |G| + |H| + 3$. En appliquant l'hypothèse d'induction on trouve donc $|F| = (4b(G) + n(G) + 1) + (4b(H) + n(H) + 1) + 3 = 4b(F) + n(F) + 1$. La relation (R) est encore vérifiée.

21. On définit en Caml le type expression logique de la façon suivante :

```
type proposition = | Var  of string
                   | Non  of expression
                   | Et   of expression * expression
                   | Ou   of expression * expression
                   | Impl of expression * expression
    ;;

 let exemple = Et(Var "x",Ou(Var "y",Var "z"));;
```

1° Écrire une fonction `variables : proposition -> string list` qui donne la liste (sans répétition) des variables présentes dans la proposition logique passée en entrée. Voici un exemple d'utilisation de la fonction `variables` :

```
#variables (Ou (Et (Var "a", Var "b"), Var "a"));;
- : string list = ["a"; "b"]
```

2° Montrer par induction que la longueur d'une proposition F qui comporte $b(F)$ occurrences de symboles de connecteurs binaires et $n(F)$ occurrences du symbole de négation est $|F| = 4b(F) + n(F) + 1$.

Solution proposée

1°

```
let variables expr_analysee =
  let rec var_aux expr liste =
     match expr with
        | Vrai -> liste
        | Faux -> liste
        | Non(e) -> var_aux e liste
        | Et(e1,e2) -> var_aux e1 (var_aux e2 liste)
        | Ou(e1,e2) -> var_aux e1 (var_aux e2 liste)
        | Imp(e1,e2) -> var_aux e1 (var_aux e2 liste)
        | Equ(e1,e2) -> var_aux e1 (var_aux e2 liste)
        | Var s -> if (mem s liste) then liste
                                    else s::liste
  in var_aux expr_analysee [];;
```

2° Longueur d'une proposition en fonction du nombre de connecteurs.
Rappelons que $b(F)$ et $n(F)$ désignent respectivement le nombre d'occurrences de symboles de connecteurs binaires et le nombre d'occurrences du symbole de négation dans l'expression de F. On va montrer par induction que :

$$|F| = 4b(F) + n(F) + 1 \qquad (R)$$

- Si $F \in \mathcal{V}$ c'est-à-dire si F est une variable propositionnelle, alors $b(F) = n(F) = 0$ et $|F| = 1$ et la relation (R) est vérifiée.
- Si $F = \neg G$ avec $|G| = 4b(G) + n(G) + 1$ (hypothèse d'induction) alors $b(F) = b(G)$, $n(F) = n(G) + 1$, $|F| = |G| + 1$ et en appliquant l'hypothèse d'induction on trouve $|F| = |G| + 1 = 4b(G) + n(G) + 1 + 1 = 4b(F) + n(F) + 1$. La relation (R) est à nouveau vérifiée.
- Si $F = (G \top H)$ où $\top$ désigne un connecteur binaire quelconque, avec $|G| = 4b(G)+n(G)+1$ et $|H| = 4b(H)+n(H)+1$ (hypothèse d'induction), alors $b(F) = b(G) + b(H) + 1$, $n(F) = n(G) + n(H)$ et $|F| = |G| + |H| + 3$. En appliquant l'hypothèse d'induction on trouve donc $|F| = (4b(G) + n(G) + 1) + (4b(H) + n(H) + 1) + 3 = 4b(F) + n(F) + 1$. La relation (R) est encore vérifiée.

22. Mots de Lukasiewicz.