Chapitre 2

Récursivité

1 Généralités

1.1 Introduction par l'exemple

Considérons une suite (u_n) définie par :

$$u_0 \in I \quad \text{et} \quad \forall n \in \mathbb{N}\,,\, u_{n+1} = f(u_n)\,,$$

ou plutôt en informatique :

$$u_0 \in I \quad \text{et} \quad \forall n \in \mathbb{N}^*\,,\, u_n = f(u_{n-1})\,,$$

où I désigne un intervalle de $\mathbb{R}$ contenant au moins deux points distincts et f une application de I dans I. Étant donné un entier naturel n, on cherche à calculer la valeur de u_n.

Approche itérative.

Pour calculer u_n, deux cas possibles :

- soit $n = 0$ et la valeur attendue est u_0,
- soit n est non nul, nous commençons avec u_0, nous calculons ensuite u_1, puis u_2, ..., puis u_{n-1}, puis u_n.

Écrivons l'algorithme en pseudo-langage :

```
suite_iteratif(u0 ∈ ℝ , n ∈ ℕ ↦ u ∈ ℝ)
     si ( n = 0 )
     alors
          u <- u0
     sinon
          u <- u0
          pour k allant de 1 à n faire
               u <- f(u)
```

Le cas général pouvant inclure le cas particulier, on donne la version simplifiée :

```
suite_iteratif(u0 ∈ ℝ , n ∈ ℕ ↦ u ∈ ℝ)
      u <- u0
      pour k allant de 1 à n faire
            u <- f(u)
```

Écrivons le programme en OCaml :

```
let suite_iteratif u0 n =
    let u = ref u0 in
        for k = 1 to n do
            u := f !u
        done;
        !u
;;
```

Approche récursive.

Pour calculer u_n, deux cas possibles :

- soit $n = 0$ et la valeur attendue est u_0,
- soit n est non nul, nous demandons le calcul de u_{n-1} et avec cette valeur de u_{n-1}, nous pourrons calculer u_n.

Écrivons l'algorithme en pseudo-langage :

```
suite_recursif(u0 ∈ ℝ , n ∈ ℕ ↦ u ∈ ℝ)
      si ( n = 0 )
      alors
            u <- u0
      sinon
            suite_recursif(u0 , n-1 ↦ u )
            u <- f(u)
```

Écrivons le programme en OCaml :

```
let rec suite_recursif u0 n =
    if (n = 0) then
        u0
    else
        f (suite_recursif u0 (n-1))
;;
```

Remarque. L'approche récursive présente deux avantages :

1. les cas particuliers apparaissent naturellement,
2. en récursif, il suffit d'exprimer le rang n en fonction du rang $n-1$ alors qu'en itératif, on doit faire une boucle de 1 à n et exprimer le rang k en fonction du rang $k-1$.

1.2 Mise en place d'un algorithme récursif

Définition.

Un algorithme est dit **récursif** lorsqu'il intervient lui-même (directement ou indirectement) dans sa définition.

Très souvent, un algorithme récursif est lié à la relation de récurrence permettant de solutionner le problème à l'aide d'une solution au même problème mais sur des données de taille strictement inférieure.

Exemples.

1. Factorielle.

   - Définition itérative de la factorielle d'un entier naturel :

   $$\begin{cases} 0! = 1 \\ \forall n \in \mathbb{N}^*,\ n! = \displaystyle\prod_{k=1}^{n} k \end{cases}$$

   ce qui nous conduit à l'algorithme itératif suivant, schéma classique d'un calcul de produit :

   – En pseudo-langage :

   ```
   fact_iter(n ∈ ℕ ↦ f ∈ ℕ)
        f <- 1
        pour k allant de 1 à n faire
             f <- f * k
   ```

   – En OCaml :

   ```
   let fact_iter n =
       let f = ref 1 in
           for k =1 to n do
               f := !f * k
           done;
       !f
   ;;
   ```

   - Définition récursive de la factorielle d'un entier naturel :

   $$\begin{cases} 0! = 1 \\ \forall n \in \mathbb{N}^*,\ n! = n \times (n-1)! \end{cases}$$

   ce qui nous conduit à l'algorithme récursif suivant :

   – En pseudo-langage :

   ```
   fact_rec(n ∈ ℕ ↦ f ∈ ℕ)
        si ( n = 0 )
        alors
             f <- 1
        sinon
             fact_rec(n-1 ↦ f )
             f <- n * f
   ```

   – En OCaml :

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

2. Puissance entière positive d'un entier relatif.

   - Définition itérative de la puissance entière positive d'un entier relatif :

   $$\begin{cases} a^0 = 1 \\ \forall n \in \mathbb{N}^*,\ a^n = \displaystyle\prod_{k=1}^{n} a \end{cases}$$

   ce qui nous conduit à l'algorithme itératif suivant :

– En pseudo-langage :

```
puissance_iter(a ∈ ℤ, n ∈ ℕ ↦ p ∈ ℤ)
     p <- 1
     pour k allant de 1 à n faire
          p <- a * p
```

– En OCaml :

```
let puissance_iter a n =
    let p = ref 1 in
        for k =1 to n do
            p := !p * a
        done;
    !p
;;
```

- Définition récursive de la puissance entière positive d'un entier relatif :

$$\begin{cases} a^0 = 1 \\ \forall n \in \mathbb{N}^*,\ a^n = a \times a^{n-1} \end{cases}$$

ce qui nous conduit à l'algorithme récursif suivant :

– En pseudo-langage :

```
puissance_rec(a ∈ ℤ, n ∈ ℕ ↦ p ∈ ℤ)
     si ( n = 0 )
     alors
          p <- 1
     sinon
          puissance_rec(a ∈ ℤ, n-1 ∈ ℕ ↦ p ∈ ℤ)
          p <- a * p
```

– En OCaml :

```
let rec puissance_rec a n =
    if (n = 0)
        then
            1
        else
            a * (puissance_rec a (n-1))
;;
```

Remarque. Dans la mise en place d'un algorithme récursif, nous aurons toujours deux soucis en tête :

1. Quel est le cas de base (qui est souvent le cas simple) ?
2. Comment solutionner mon problème si je sais le solutionner sur des données strictement inférieures ?

1.3 Empilement et dépilement des appels

Nous avons un mode sur OCaml qui nous permet de visualiser les appels récursifs d'une fonction, le mode `trace` (c'est-à-dire exécution pas à pas). En reprenant l'exemple de la fonction récursive factorielle :

```
#trace fact_rec ;;
fact_rec is now traced.
```

Regardons comment est évaluée l'expression `fact_rec 5` par OCaml :

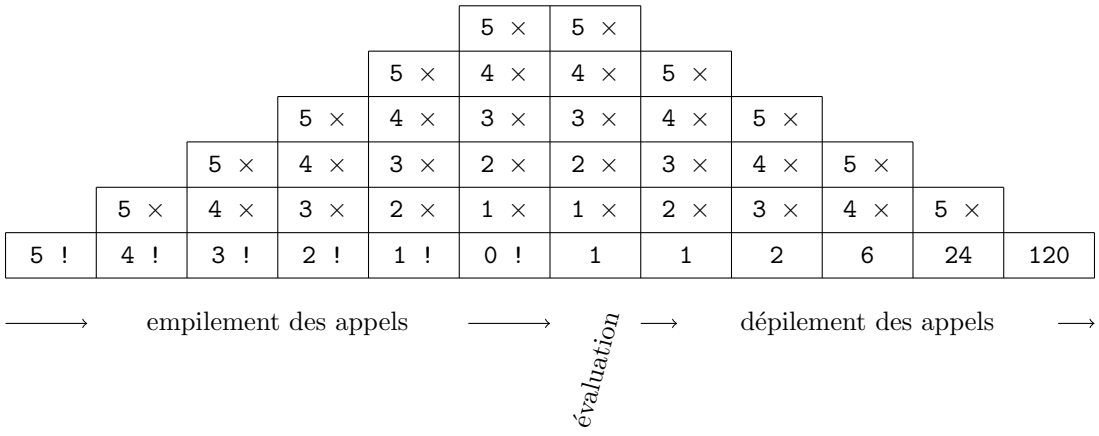
```
fact_rec 5;;
fact_rec <-- 5
fact_rec <-- 4
fact_rec <-- 3
fact_rec <-- 2
fact_rec <-- 1
fact_rec <-- 0
fact_rec --> 1
fact_rec --> 1
fact_rec --> 2
fact_rec --> 6
fact_rec --> 24
fact_rec --> 120
- : int = 120

#untrace fact_rec;;
fact_rec is no longer traced.
```

Lorsqu'on souhaite calculer `fact_rec 5` :

- Cet appel renvoie dans un premier temps `5*fact_rec(4)`, il est donc mémorisé le fait que 5 est à multiplier par un résultat non encore disponible. On dit qu'il y a un **empilement du contexte**.
- A son tour, l'expression `fact_rec(4)` est évaluée en `4*fact_rec(3)` et le contexte est empilé.
- Puis l'expression `fact_rec(3)` est évaluée en `3*fact_rec(2)` et le contexte est empilé.
- L'expression `fact_rec(2)` est évaluée en `2*fact_rec(1)` et le contexte est empilé.
- L'expression `fact_rec(1)` est évaluée en `1*fact_rec(0)` et le contexte est empilé.
- Enfin `fact_rec(0)` est évaluée en 1 : il s'agit du **cas de base** , c'est-à-dire une expression pour laquelle l'expression est évaluée directement, sans faire appel à elle-même.
- Mais ce n'est pas encore fini ! Il faut à présent procéder au **dépilement du contexte**.

Le diagramme suivant schématise l'ensemble du mécanisme :



Remarque. Ce processus d'empilement du contexte nécessite en général pas mal de place en mémoire, un des inconvénients de la programmation récursive qui l'a longtemps pénalisée.
Notamment, si la fonction est mal définie et s'appelle indéfiniment, un message d'erreur du genre `stack overflow` ou `out of memory` finit par apparaître.

2 Différents modes de récursivité

2.1 La récursivité simple

Nous parlons de **récursivité simple** lorsque la fonction fait un seul appel récursif à elle-même.

Exemples.

1. Écrire une fonction récursive qui calcule le carré d'un entier naturel n, en remarquant la relation de récurrence suivante :

$$n^2 = (n-1+1)^2 = (n-1)^2 + 2(n-1) + 1 = (n-1)^2 + 2n - 1.$$

```
let rec carre n = match n with
    | 0 -> 0
    | _ -> carre (n-1) + 2*n - 1
;;
```

2. Écrire une fonction récursive qui calcule la somme de deux entiers naturels (p, q) :

$$p + q = \begin{cases} p & \text{si } q = 0, \\ (p+1) + (q-1) & \text{si } q \geq 1. \end{cases}.$$

```
let rec somme p q = match q with
    | 0 -> p
    | _ -> somme (p+1) (q-1)
;;
```

3. Écrire une fonction récursive qui calcule le produit de deux entiers naturels (p, q) :

$$p \times q = \begin{cases} 0 & \text{si } q = 0, \\ p \times (q-1) + p & \text{si } q \geq 1. \end{cases}.$$

```
let rec produit p q = match q with
    | 0 -> 0
    | _ -> (produit p (q-1))+p
;;
```

4. Écrire une fonction récursive qui calcule le minimum des éléments d'un tableau t.

   Première version : Algorithme "naïf".

   Soit `n = Array.length t`

   - Cas de base : si `n = 1` alors le minimum vaut `t.(0)`
   - Cas général : Nous demandons le calcul du minimum de `Array.sub t 1 (n-1)`, noté `m`, et nous comparons `m` avec `t.(0)` et renvoyons le plus petit des deux.

```
let rec min1 t =
    let n = Array.length t in
        if (n = 1) then
            t.(0)
        else
            let m = min1 (Array.sub t 1 (n-1)) in
                if (t.(0) < m) then
                    t.(0)
                else
                    m
;;
```

Il est toujours plus intéressant d'avoir l'emplacement du minimum dans le tableau (indice du minimum au lieu de la valeur du minimum). Modifions légèrement le programme précédent :

```
let rec min1 t =
    let n = Array.length t in
        if (n = 1) then
            0
        else
            let i = 1 + min1 (Array.sub t 1 (n-1)) in
                if (t.(0) < t.(i)) then
                    0
                else
                    i
;;
```

Deuxième version : Algorithme en travaillant en place (avec un pointeur).

Nous définissons une fonction auxiliaire qui détermine l'emplacement du minimum des valeurs du tableau à partir de l'emplacement numéro i (pointeur) :

```
let rec min_aux t i =
    let n = Array.length t in
        if (i = n-1) then
           i
        else
            let m = min_aux t (i+1) in
                if (t.(i) < t.(m)) then
                    i
                else
                    m
;;
```

Nous pouvons alors définir la fonction minimum ainsi :

```
let min2 t =
    min_aux t 0
;;
```

L'avantage de cette deuxième version est de travailler en place. L'algorithme `min2` travaille toujours sur le même tableau, en déplaçant uniquement un curseur/pointeur (l'indice i), tandis que chaque appel récursif de la fonction `min1` nécessitait de créer un nouveau tableau (sous-tableau) ce qui engendre une mauvaise complexité spatiale.

5. Écrire une fonction récursive `min_max` qui calcule un indice de la valeur minimale et un indice de la valeur maximale des valeurs contenues dans un tableau t.

```
let rec min_max_aux t i =
    let n = Array.length t in
        if (i = n-1) then
            (i,i)
        else
            let (mi, ma) = min_max_aux t (i+1) in
                match i with
                    | i when t.(i) < t.(mi) -> (i,ma)
                    | i when t.(i) > t.(ma) -> (mi,i)
                    | _ -> (mi, ma)
;;

let min_max t =
   min_max_aux t 0
;;
```

Remarque. Comme le montre les trois exemples qui suivent, la récursivité ne fait pas toujours appel à des données de taille immédiatement inférieure.

6. Écrire une fonction récursive qui calcule le produit de deux entiers naturels (p, q) par l'algorithme égyptien :

$$p \times q = \begin{cases} (2 * p) * (q/2) & \text{si } q \text{ est pair,} \\ p \times (q-1) + p & \text{si } q \text{ est impair.} \end{cases}$$

```
let rec produit_egyptien p q = match q with
    | 0 -> 0
    | _ -> match (q mod 2) with
                | 0 -> produit_egyptien (2*p) (q/2)
                | 1 -> (produit_egyptien p (q-1))+p
;;
```

7. Écrire une fonction récursive qui calcule le quotient et le reste d'une division euclidienne dans $\mathbb{N}$, en utilisant que :

$$a = qb + r \Leftrightarrow a - b = (q-1)b + r.$$

```
let rec division_euclidienne a b =
    if (a < b)
        then
            (0,a)
        else
            let q,r = division_euclidienne (a-b) b in
                (q+1,r)
;;
```

8. Écrire une fonction récursive qui calcule le pgcd de deux entiers naturels (a, b).

   Première version : En singeant l'algorithme d'Euclide.

```
let rec pgcd1 a b =
    if (b = 0)
        then
            a
        else
            let r = snd (division_euclidienne a b) in
                pgcd1 b r
;;
```

   Deuxième version : En utilisant les propriétés du pgcd.

   Rappelons que :

$$\begin{cases} a \wedge 0 &= a \\ a \wedge b &= b \wedge a \\ a \wedge b &= (a-b) \wedge b \end{cases}$$

   $24 \wedge 10 =$

   Nous pouvons alors utiliser l'algorithme suivant qui est strictement décroissant selon l'ordre lexicographique sur le couple (b, a) :

   - Cas de base : si $b = 0$ alors on renvoie a.
   - Cas général :
     - si $a \geq b$, nous demandons le pgcd de $a - b$ et b.
     - si $a < b$, nous demandons le pgcd de b et a.

```
let rec pgcd2 a b =
  if (b = 0)
    then
      a
    else
      if (a >= b)
         then
           pgcd2 (a-b) b
         else
           pgcd2 b a
;;
```

Notons que cette dernière récursivité travaille à espace mémoire constant.

2.2 La récursivité double ou multiple

Nous parlons de **récursivité double** (ou **multiple**) lorsque la fonction fait deux (ou plusieurs) appels récursifs à elle-même.

Exemples.

1. Considérons la suite (F_n) dite de Fibonacci, définie par :

$$F_0 = 0, \quad F_1 = 1 \quad \text{et} \quad \forall n \in \mathbb{N}\,,\ F_{n+2} = F_{n+1} + F_n\,.$$

Écrire une fonction récursive qui calcule pour un entier naturel n, la valeur de F_n.

```
let rec fibonacci n = match n with
    | 0 -> 0
    | 1 -> 1
    | _ -> fibonacci (n-1) + fibonacci (n-2)
;;
```

2. Écrire une fonction récursive qui calcule les coefficients binomiaux.

Première version : A l'aide la formule de Pascal.

Rappelons la formule du triangle de Pascal :

$$\forall n \in \mathbb{N}^*,\ \forall p \in [\![1, n-1]\!],\ \binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}.$$

```
let rec coefficient_binomial1 n p = match (n,p) with
    | (n,0) -> 1
    | (n,p) when (p > n) -> 0
    | _ ->(coefficient_binomial1 (n-1) p) + (coefficient_binomial1 (n-1) (p-1))
;;
```

Deuxième version : A l'aide de la formule diagonale.

Rappelons la formule diagonale :

$$\forall n \in \mathbb{N}^*,\ \forall p \in [\![1, n]\!],\ \binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}.$$

```
let rec coefficient_binomial2 n p =
    if (p = 0)
        then
            1
        else
            (coefficient_binomial2 (n-1) (p-1)) * n/p
;;
```

3. Les tours de Hanoï.

   Soit n un entier naturel non nul. Nous avons n disques empilés sur la structure 1, tous de tailles différentes et empilés par ordre de taille décroissant.

   Notre but est de déplacer les disques de la structure 1 à la structure 3 en manipulant 1 disque à la fois sans jamais poser un plus grand sur un plus petit.

   - Traiter les exemples avec 1, 2, 3, 4 disques.

   - Écrire un algorithme récursif permettant d'afficher les déplacements nécessaires pour déplacer les n disques suivant la règle imposée.
     - En pseudo-langage :
       Soient $i, j, k \in \{1; 2; 3\}$ deux à deux distincts et $n \in \mathbb{N}^*$. Écrire en pseudo-langage une fonction récursive `deplace` d'arguments n le nombre de disques à déplacer, i la structure de départ, j la structure d'arrivée et k la structure intermédiaire.

       ```
       deplace(n,i,j,k ∈ ℕ -> )
             si n = 1
                  alors
                       Afficher "i -> j"
                  sinon
                       deplace(n-1,i,k,j)
                       Afficher "i -> j"
                       deplace(n-1,k,j,i)
       ```

       En déduire une fonction `hanoi` répondant au problème.

       ```
       hanoi(n ∈ ℕ -> )
             deplace(n,1,3,2)
       ```

     - En OCaml :

       ```
       let rec deplace n i j k =
           if n = 1
               then
                   print_string (string_of_int i ^ "->" ^ string_of_int j)
               else
                   begin
                       deplace (n-1) i k j;
                       print_string (string_of_int i ^ "->" ^ string_of_int j);
                       deplace (n-1) k j i
                   end
       ;;

       let hanoi n =
           deplace n 1 3 2
       ;;
       ```

2.3 La récursivité croisée

Nous parlons de **récursivité croisée** pour deux fonctions récursives telles que chacune fait appel à l'autre.

1. Considérons les suites (a_n) et (b_n) définies par :

$$a_0 \in \mathbb{R}_+, \quad b_0 \in \mathbb{R}_+, \quad \text{et} \quad \forall n \in \mathbb{N}, \quad a_{n+1} = \frac{a_n + b_n}{2}, \quad b_{n+1} = \sqrt{a_n \times b_n}.$$

   Écrire une fonction récursive qui calcule pour un entier naturel n, les valeurs de a_n et b_n.

```
let rec suitea n a0 b0 =
    if (n = 0)
        then
            a0
        else
            ((suitea (n-1) a0 b0) +. (suiteb (n-1) a0 b0)) /. 2.0
and suiteb n a0 b0 =
    if (n = 0)
        then
            b0
        else
            sqrt((suitea (n-1) a0 b0) *. (suiteb (n-1) a0 b0))
;;
```

2. Écrire deux fonctions récursives `pair` et `impair` qui pour un entier naturel n, la première retourne `true` si et seulement si n est pair, la seconde retourne `true` si et seulement si n est impair.

```
let rec pair n = match n with
    | 0 -> true
    | _ -> impair (n-1)
and impair n = match n with
    | 0 -> false
    | _ -> pair (n-1)
;;
```

2.4 La récursivité terminale

Parmi les fonctions récursives, il existe une sous-classe : celle des fonctions à récursivité dite terminale.

Définition.

> Une fonction est dite **récursive terminale** quand la valeur retournée est directement la valeur obtenue par l'appel récursif, sans qu'il n'y ait aucune opération sur cette valeur.

Exemples.

1. Le calcul récursif du pgcd de deux entiers naturels (fonction `pgcd2`) est une récursivité terminale.
2. Le calcul récursif de la factorielle d'un entier naturel est une récursivité non terminale.

Remarques.

- Dans le cas d'une fonction récursive terminale, il n'y a plus pour OCaml qu'à calculer la valeur pour cet appel et il n'y a donc pas d'empilement. La complexité spatiale d'une telle fonction est donc réduite (comme dans le cas d'une programmation itérative).
- L'intérêt est que les fonctions récursives terminales sont plus efficaces : elles s'évaluent avec espace mémoire constant.
- On ne risque pas dans ce cas de dépasser la taille limite de la pile d'exécution ("`stack overflow`").

Il y a toute légitimité à vouloir privilégier les fonctions récursives terminales, qui constituent souvent le juste milieu entre la clarté des fonctions récursives, et la rapidité supposée de la programmation impérative.

Méthode.

Pour rendre terminales certaines fonctions récursives non terminales, on utilise un (ou plusieurs) ***accumulateur(s)*** *qui permet de faire passer les résultats entre les appels récursifs.*

Exemples.

1. Calcul de la factorielle d'un entier naturel.

   Première version :

```
(*fonction auxiliaire*)

let rec fact_aux1 n acc k =
    if (k = n)
        then
            acc
        else
            fact_aux1 n (acc*(k+1)) (k+1)
;;

(*fonction chapeau*)

let fact_terminale1 n =
    fact_aux1 n 1 0
;;
```

   Lorsqu'on évalue l'expression `facto_terminale1 5` sur OCaml, on aura alors la suite d'appels de la fonction `facto_aux1` suivante :

```
->facto_aux1 5 1 0
->facto_aux1 5 1 1
->facto_aux1 5 2 2
->facto_aux1 5 6 3
->facto_aux1 5 24 4
->facto_aux1 5 120 5
->120
```

   Il y a donc empilement puis évaluation mais pas de dépilement.

   Deuxième version : partir de n pour aller à 0 au lieu de partir de 0 pour aller à n.

```
(*fonction auxiliaire*)

let rec facto_aux2 n acc =
    if (n = 0)
        then
            acc
        else
            facto_aux2 (n-1) (acc*n)
;;

(*fonction chapeau*)

let facto_terminale2 n =
    facto_aux2 n 1
;;
```

2. Calcul de la puissance entière positive d'un entier relatif.
   Première version :

```
(*fonction auxiliaire*)

let rec puissance_aux1 a n acc k =
    if (k = n)
        then
            acc
        else
            puissance_aux1 a n (acc*a) (k+1)
;;

(*fonction chapeau*)

let puissance_terminale1 a n =
    puissance_aux1 a n 1 0
;;
```

   Deuxième version :

```
(*fonction auxiliaire*)

let rec puissance_aux2 a n acc =
    if (n = 0)
        then
            acc
        else
            puissance_aux2 a (n-1) (acc*a)
;;

(*fonction chapeau*)

let puissance_terminale2 a n =
    puissance_aux2 a n 1
;;
```

3. Calcul des termes successifs d'une suite récurrente du type $u_{n+1} = f(u_n)$.
   Première version :

```
(*fonction auxiliaire*)

let rec suite_aux1 n f acc k =
    if (k = n)
        then
            acc
        else
            suite_aux1 n f (f acc) (k+1)
;;

(*fonction chapeau*)

let suite_terminale1 n f u0 =
    suite_aux1 n f u0 0
;;
```

Deuxième version :

```
let rec suite_terminale2 n f u0 =
    if (n = 0)
        then
            u0
        else
            suite_terminale2 (n-1) f (f u0)
;;
```

4. Calcul de la somme des carrés des inverses des premiers entiers.

```
(*fonction auxiliaire*)

let rec somme_inv_aux n acc =
    if (n = 1.0)
        then
            acc
        else
            somme_inv_aux (n -. 1.0) (acc +. (1.0 /. n)**2)
;;

(*fonction chapeau*)

let somme_inv n =
    somme_inv_aux (float_of_int n) 1.0
;;
```

5. Calcul des termes successifs de la suite de Fibonacci.

```
(*fonction auxiliaire*)

let rec fibonacci_aux n acc1 acc2 =
    if (n = 0)
        then
            acc1
        else
            begin
                if (n = 1)
                    then
                        acc2
                    else
                        fibonacci_aux (n-1) acc2 (acc1+acc2)
            end
;;

(*fonction chapeau*)

let fibonacci_terminale n =
    fibonacci_aux n 0 1
;;
```

3 Itération et récursivité

3.1 De l'itératif au récursif

Si d est l'ensemble des données manipulées dans le corps d'une boucle, d_0 l'état initial de ces données et φ la transformation qui affecte ces données dans le corps de la boucle, effectuer cette boucle revient à itérer la suite définie par d_0 et la relation : $d_{n+1} = \varphi(d_n)$.

Exemples.

- Calcul de la puissance d'un entier relatif :
$$\forall n \in \mathbb{N},\ d_n = a^n \text{ donc } d_0 = 1 \text{ et } \varphi : x \mapsto ax.$$
- Calcul de la factorielle d'un entier naturel :
$$\forall n \in \mathbb{N},\ d_n = (n, n!) \text{ donc } d_0 = (0,1) \text{ et } \varphi : (x,y) \mapsto (x+1, (x+1)y).$$

Cas des boucles indexées.

Nous pouvons proposer la construction d'un itérateur générique :

```
let rec pour phi d n = match n with
    | 0 -> d
    | _ -> phi (pour phi d (n-1))
;;
```

ou en version terminale :

```
let rec pour phi d n = match n with
    | 0 -> d
    | _ -> pour phi (phi d) (n-1)
;;
```

Son type est : `('a -> 'a) -> 'a -> int -> 'a`.

Les fonctions précédentes peuvent alors s'écrire :

```
let puissance a n = pour (function x -> a*x) 1 n
;;
```

```
let fact n = snd (pour (function (x,y) -> (x+1,(x+1)*y)) (0,1) n)
;;
```

Comme la fonction `pour` est récursive, on en déduit la :

Propriété 1

Tout algorithme utilisant une boucle indexée possède une version récursive.

Pour obtenir la version récursive d'un algorithme utilisant une boucle indexée, il suffit donc de déterminer l'itérateur associé à cette boucle. Par exemple, on obtiendra pour les deux fonctions précédentes :

```
let rec puissance a n = match n with
    | 0 -> 1
    | _ -> a * puissance a (n-1)
;;
```

```
let rec fact n = match n with
    | 0 -> 1
    | _ -> n * fact (n-1)
;;
```

Cas des boucles conditionnelles.

Nous pouvons écrire une fonction `tant_que` générique de la manière suivante :

```
let rec tant_que cond phi d = match (cond d) with
    | true -> tant_que cond phi (phi d)
    | false -> d
;;
```

Il s'agit d'une fonction de type : `('a -> bool) -> ('a -> 'a) -> 'a -> 'a`.

Elle retourne le premier élément de la suite $(d_n)_{n\in\mathbb{N}}$ définie par $d_0 = d$ et $d_{n+1} = \varphi(d_n)$ ne vérifiant pas la condition.

Comme la fonction `tant_que` est récursive terminale, on en déduit la :

Propriété 2

Tout algorithme utilisant une boucle conditionnelle possède une version récursive.

Grâce aux deux dernières propriétés, on a finalement le :

Théorème 3

Tout algorithme itératif possède une version récursive.

3.2 Du récursif au récursif terminal

La fonction récursive `pour` a pour objet de calculer $d_n = (\varphi \circ \varphi \circ \ldots \circ \varphi)(d_0)$. Les deux versions proposées précédemment sont différentes puisque la seconde est récursive terminale, au contraire de la première :

- dans sa première version (non terminale), d_n est calculé par l'intermédiaire de la relation : $\varphi^n = \varphi \circ \varphi^{n-1}$,
- dans la seconde (terminale), d_n est calculé par : $\varphi^n = \varphi^{n-1} \circ \varphi$.

De cette manière, nous disposons d'une méthode pour rendre terminal un algorithme récursif qui ne l'est pas : il suffit de chercher l'itérateur correspondant, et de suivre alors le modèle donné.

Nous avons donc le :

Théorème 4

Tout algorithme récursif possède une version récursive terminale.

Donnons par exemple une version récursive terminale de la fonction factorielle suivante :

```
let rec fact n = match n with
    | 0 -> 1
    | _ -> n * fact (n-1)
;;
```

Nous avons vu que l'itérateur associé est la suite $(d_n = (n, n!))$ définie par la donnée de $d_0 = (0, 1)$ et la relation $d_{n+1} = \varphi(d_n)$ avec $\varphi : (x, y) \mapsto (x + 1, (x + 1)y)$.

En suivant le modèle de la fonction `pour` donné ci-dessus, on obtient :

```
(*fonction auxiliaire*)

let rec aux (x,y) n = match n with
    | 0 -> y
    | _ -> aux (x+1,(x+1)*y) (n-1)
;;

(*fonction chapeau*)

let fact n =
    aux (0,1) n
;;
```

qui est maintenant récursive terminale.

De la même manière, donner une versions récursives terminales des fonctions :

1. Permettant de calculer a^n :

```
(*fonction auxiliaire*)

let rec aux a x n = match n with
    | 0 -> x
    | _ -> aux a (a*x) (n-1)
;;

(*fonction chapeau*)

let puissance a n =
    aux a 1 n
;;
```

2. Permettant de calculer la valeur F_n du n-ième terme de la suite de Fibonacci :

```
(*fonction auxiliaire*)

let rec aux (x,y) n = match n with
    | 0 -> y
    | n -> aux (x+y,x) (n-1)
;;

(*fonction chapeau*)

let fibonacci n =
    aux (1,0) n
;;
```

3.3 Du récursif terminal à l'itératif

Une fonction récursive terminale se transforme facilement en version itérative (en notant toujours φ l'itérateur) :

- Version récursive :

```
let rec f x = match x with
    | x when cond x -> g x
    | _ -> f (phi x)
;;
```

- Version itérative :

```
let f x =
    let y = ref x in
        while not (cond !y) do
            y := phi !y
        done;
    g !y
;;
```

En suivant ce modèle, on peut ainsi traduire itérativement n'importe quel algorithme récursif terminal. On a donc le :

Théorème 5

Tout algorithme récursif terminal possède une version itérative.

Exemple. Reprendre la fonction `pgcd2` qui est récursive terminale et la traduire itérativement.

```
let pgcd2 a b =
    let x = ref a and y = ref b in
        while !y > 0 do
            if (!x >= !y)
                then
                    x := !x - !y
                else
                    begin
                        let aux = !x in
                            x := !y ;
                            y := aux
                    end
        done ;
    !x
;;
```