Preuve de Programme

27 novembre 2023

Plan

Administratif

Rendu du DM2

Retours semaine passée

Preuve de programme

Administratif

Un peu d'organisation

Vous n'avez pas colle mardi matin, mais un DS de SI à la place. La colle est rattrapée le **mercredi 29 Novembre de 16h35 à 17h30 en B309** La colle du mardi soir n'est pas affectée.

Rendu du DM2

Notes du DM

Moyenne	Q1	Médiane	Q3	Max
15,14	11,9	15,6	18,8	20(21,6)

Les notes au dessus de 20 seront plafonnées à 20 pour le semestre.

Quelques informations supplémentaires

Il s'avère qu'il est conseillé d'apprendre à se servir des différentes bibliothèques OCaml de sorte à gagner du temps pour certaines questions, en particuliers les modules `List` et `Array`.
Nous aurons le temps de nous entraîner avec ces structures au second semestre.

Expliquez votre code

La prochaine fois que je vois un code que je ne comprends pas au bout de 60 secondes et où il n'y a pas d'explication, c'est zéro.

Expliquez votre code

La prochaine fois que je vois un code que je ne comprends pas au bout de 60 secondes et où il n'y a pas d'explication, c'est zéro. En règle général, les filtrages sont plus lisibles que des conditions imbriquées et mal indentées.

Expliquez votre code

La prochaine fois que je vois un code que je ne comprends pas au bout de 60 secondes et où il n'y a pas d'explication, c'est zéro.
En règle général, les filtrages sont plus lisibles que des conditions imbriquées et mal indentées.
Parfois, j'ai l'impression que vos codes sont très long parce que vous avez beaucoup bidouillé sur la machine sans réfléchir à comment faire un code clair.

Mise en page

Il y a des copies avec une mise en page difficile à suivre.
Essayez de fournir quelque chose au format pdf en dimension A4.
Le prochain DM sera en C, et il faudra à nouveau le rendre en manuscrit sur papier.

Utilisation des begin/end

J'ai vu de très bonnes utilisations des begin/ end dans les structures conditionnelles, c'est très agréable à lire quand tout est indenté correctement.

```
1  if (...)
2      then
3          begin
4              ...
5              ...
6          end
7      else
8          begin
9              ...
10             ...
11         end
```

Supérieur, supérieur strictement

Que veut dire a supérieur à b ?

Supérieur, supérieur strictement

Que veut dire a supérieur à b ?

$$a \geqslant b$$

Supérieur, supérieur strictement

Que veut dire a supérieur à b ?

$$a \geqslant b$$

Que veut dire a strictement supérieur à b ?

Supérieur, supérieur strictement

Que veut dire a supérieur à b ?

$$a \geqslant b$$

Que veut dire a strictement supérieur à b ?

$$a > b$$

Programmation de la fonction puissance

```
1 let puissance n k =
2     let fn = float_of_int n in
3     let fk = float_of_int k in
4     int_of_float (n ** k)
```

Programmation de la fonction puissance

```
let puissance n k =
    let fn = float_of_int n in
    let fk = float_of_int k in
    int_of_float (n ** k)
```

Bien :

```
let rec puissance n = function
| 0 -> 1
| k -> n * (puissance n (k-1))
```

Programmation de la fonction puissance

```
let puissance n k =
    let fn = float_of_int n in
    let fk = float_of_int k in
    int_of_float (n ** k)
```

Bien :

```
let rec puissance n = function
| 0 -> 1
| k -> n * (puissance n (k-1))
```

Mieux :

```
let puissance n k =
    let rec aux acc = function
    | 0 -> acc
    | k -> aux (acc * n) (k-1)
    in aux 1 k
```

On peut faire un peu mieux

```
let vers_entier t =
    let rec aux k facteur acc = match k with
        |-1 -> acc
        | _ ->
            let terme =
                if t.(k) = 1
                    then facteur
                    else 0
            in aux (k-1) (facteur * 2) (acc + terme)
    in
    aux (taille_entier - 1) 1 0
```

Explications

En règle générale, vous expliquez mal les principaux changement à faire pour adapter un code : il ne suffit pas de remplacer simplement les entiers par les booléens, mais changer aussi les opérations que vous réalisez dessus.

Explications

En règle générale, vous expliquez mal les principaux changement à faire pour adapter un code : il ne suffit pas de remplacer simplement les entiers par les booléens, mais changer aussi les opérations que vous réalisez dessus.
C'est un exercice difficile que d'identifier les principales différences ou difficultés d'un programme.

Représentation des entiers

Nous reverrons la représentation des entiers en C la semaine prochaine pour les détails d'implémentation.

Retours semaine passée

Complexité amortie d'une file avec deux piles

Preuve de programme

Terminaison

Définition 1 : Terminaison

On dit qu'un programme **termine** si, quelque soit l'entrée, le programme est calculé en temps fini.

Essentiellement, il faut qu'il n'y ait pas de boucles inifinies ou d'appels récursifs infinis, *quel que soit l'entrée*.

Terminaison

Définition 1 : Terminaison

On dit qu'un programme **termine** si, quelque soit l'entrée, le programme est calculé en temps fini.

Essentiellement, il faut qu'il n'y ait pas de boucles inifinies ou d'appels récursifs infinis, *quel que soit l'entrée*.
Notre principale outil est la propriété suivante :

Proposition 1

Il n'existe pas de suite strictement décroissante infinie dans $\mathbb{N}$.

Terminaison

Définition 1 : Terminaison

On dit qu'un programme **termine** si, quelque soit l'entrée, le programme est calculé en temps fini.

Essentiellement, il faut qu'il n'y ait pas de boucles inifinies ou d'appels récursifs infinis, *quel que soit l'entrée*.
Notre principale outil est la propriété suivante :

Proposition 1

Il n'existe pas de suite strictement décroissante infinie dans $\mathbb{N}$.

Demonstration : La preuve repose sur le fait que toute partie non-vide de $\mathbb{N}$ admet un plus petit élément. Si on considère l'ensemble des valeurs d'une telle suite supposée existante, on aboutit à une contradiction.

Variant

Ainsi, la plupart du temps, pour montrer qu'une boucle termine, ou qu'il n'y a pas d'appels récursifs infinis, on cherche à exhiber un *variant de boucle*, c'est-à-dire une grandeur entière positive qui décroit strictement à chaque appel ou à chaque tour de boucle.

Exemple 1, une fonction récursive

```
1 let rec pgdc a b = match a with
2 | 0 -> b
3 | _ -> pgcd (b mod a) a
```

Exemple 1, une fonction récursive

```
1 let rec pgdc a b = match a with
2 | 0 -> b
3 | _ -> pgcd (b mod a) a
```

La suite des valeurs de *a* lors des différents appels est une suite strictement décroissante, et il s'agit d'entiers positifs : cette fonction *termine*.

Exemple 2, une boucle

```
let recherche t x =
    let n = Array.length t in
    let rec aux i j =
        if i>j then false
        else let k = (i+j)/2 in
            if x = t.(k) then true
            else
            if x> t.(k) then aux (k+1) j
            else aux i (k-1)
    in aux 0 (n-1)
```

Exemple 2, une boucle

```
1  let recherche t x =
2      let n = Array.length t in
3      let rec aux i j =
4          if i>j then false
5          else let k = (i+j)/2 in
6              if x = t.(k) then true
7              else
8              if x> t.(k) then aux (k+1) j
9              else aux i (k-1)
10     in aux 0 (n-1)
```

La grandeur $j - i + 1$ est une quantité positive qui décroit strictement à chaque appel.

Exemple 3

On propose le programme suivant qui renvoie deux entiers aléatoires distincts entre 0 et n :

```
let () = Random.init ()
let rec paire_aleatoire n =
    let a = Random.int n in
    let b = Random.int n in
    if a = b
        then paire_aleatoire n
        else a, b
```

Exemple 3

On propose le programme suivant qui renvoie deux entiers aléatoires distincts entre 0 et n :

```
let () = Random.init ()
let rec paire_aleatoire n =
    let a = Random.int n in
    let b = Random.int n in
    if a = b
        then paire_aleatoire n
        else a, b
```

Est-ce que ce code termine ?

Exemple 3

On propose le programme suivant qui renvoie deux entiers aléatoires distincts entre 0 et n :

```
let () = Random.init ()
let rec paire_aleatoire n =
    let a = Random.int n in
    let b = Random.int n in
    if a = b
        then paire_aleatoire n
        else a, b
```

Est-ce que ce code termine ?
Cette fonction termine avec probabilité 1, elle termine presque sûrement, mais elle ne termine pas toujours.

Exemple 3

On peut faire un peu mieux dans notre code pour pouvoir prouver que notre code se termine en temps constant.

```
let () = Random.init ()
let rec paire_aleatoire n-1 =
    let a = Random.int n in
    let b = Random.int (n-1) in
    if b >= a
        then a, b+1
        else a, b
```

Correction

Définition 2 : Correction

Un programme est dit **correcte** si, pour toute entrée, sa sortie correspond à la spécification.

Correction

Définition 2 : Correction

Un programme est dit **correcte** si, pour toute entrée, sa sortie correspond à la spécification.

La correction dépend d'une spécification.

Correction

Définition 2 : Correction

Un programme est dit **correcte** si, pour toute entrée, sa sortie correspond à la spécification.

La correction dépend d'une spécification.
Le meilleur outil que l'on a pour l'instant est le raisonnement par récurrence : on peut utiliser des propriétés invariantes à chaque tour de boucle ou à chaque appel récursif pour s'assurer de la correction de notre programme.

Exemple

```
1 let fibo n =
2     let rec aux a b k =
3         if k = n then a
4         else aux b (a+b) (k+1)
5     in aux 0 1 0
```

On remarque que la propriété suivante a lieu : à chaque appel à aux, $a = u_k$ et $b = u_{k+1}$.

Exemple

```
1 let fibo n =
2     let rec aux a b k =
3         if k = n then a
4         else aux b (a+b) (k+1)
5     in aux 0 1 0
```

On remarque que la propriété suivante a lieu : à chaque appel à aux, $a = u_k$ et $b = u_{k+1}$.
Lors du premier appel, on a $a = 0 = u_0$, $b = 1 = u_1$, et $k = 0$.

Exemple

```
1 let fibo n =
2     let rec aux a b k =
3         if k = n then a
4         else aux b (a+b) (k+1)
5     in aux 0 1 0
```

On remarque que la propriété suivante a lieu : à chaque appel à aux, $a = u_k$ et $b = u_{k+1}$.
Lors du premier appel, on a $a = 0 = u_0$, $b = 1 = u_1$, et $k = 0$.
À chaque appel récursif, si $a = u_k$ et $b = u_{k+1}$, le prochain appel se fait avec les arguments $b = u_{k+1}$, $a + b = u_k + u_{k+1} = u_{k+2}$ et $k + 1$.

Exemple

```
1 let fibo n =
2     let rec aux a b k =
3         if k = n then a
4         else aux b (a+b) (k+1)
5     in aux 0 1 0
```

On remarque que la propriété suivante a lieu : à chaque appel à aux, $a = u_k$ et $b = u_{k+1}$.
Lors du premier appel, on a $a = 0 = u_0$, $b = 1 = u_1$, et $k = 0$.
À chaque appel récursif, si $a = u_k$ et $b = u_{k+1}$, le prochain appel se fait avec les arguments $b = u_{k+1}$, $a + b = u_k + u_{k+1} = u_{k+2}$ et $k + 1$.
Par ailleurs ce programme termine puisque $n - k$ est une suite positive décroissante strictement. Donc ce programme termine et est correcte.

Précondition, Postcondition

Définition 3 : Précondition

Une **Précondition** est une propriété qui doit être vérifiée avant l'exécution d'un code donné.

Précondition, Postcondition

Définition 3 : Précondition

Une **Précondition** est une propriété qui doit être vérifiée avant l'exécution d'un code donné.

Définition 4 : Postcondition

Une **Postcondition** est une propriété qui doit être vérifiée après l'exécution d'un code donné.

Précondition, Postcondition

Définition 3 : Précondition

Une **Précondition** est une propriété qui doit être vérifiée avant l'exécution d'un code donné.

Définition 4 : Postcondition

Une **Postcondition** est une propriété qui doit être vérifiée après l'exécution d'un code donné.

Les préconditions et postconditions font partis de la spécification. Le morceau de code (souvent dans notre cas une fonction) exige que la précondition soit respecté lors de son exécution, et assure que la postcondition sera vérifié.

Exemple

```
1 let f x =
2     (* Pre condition : x>= 0.
3     Post condition : x<= 0.
4     *)
5     -. sqrt x
```

Assertion (1)

Pour s'assurer d'une propriété, on peut utiliser une fonction proposée par OCaml : `assert`.
`assert` n'est pas exactement une fonction, on doit nécessairement avoir un argument.
Si le booléen en argument du `assert` est évalué à faux, l'assertion lève une erreur, sinon, rien ne se passe.

```
1 assert (1 = 0)
```

```
1 assert (1 = 1)
```

Assertion (2)

Une assertion permet de s'assurer d'une précondition à l'entrée d'une fonction :

```
let rec pgdc a b =
assert (b >= a) ;
match a with
| 0 -> b
| _ -> pgcd (b mod a) a
```

Assertion (2)

Une assertion permet de s'assurer d'une précondition à l'entrée d'une fonction :

```
let rec pgdc a b =
assert (b >= a) ;
match a with
| 0 -> b
| _ -> pgcd (b mod a) a
```

On peut aussi s'en servir pour s'assurer qu'une sortie vérifie une postcondition.

Invariant sur une structure

Définition 5 : Invariant de structure

Un **invariant de structure** est une propriété sur une structure qui est vrai après chaque opération sur la structure.

L'invariant peut ne plus être vrai au cours d'une modification, mais doit redevenir vrai à la fin de l'opération.

Exemple

Exemple 1

Dans un tableau dynamique tel que nous l'avons implémenté la semaine dernière, `td.curseur` est toujours inférieur ou égal à la taille de `td.memoire`.

Exemple

```
let ajouter td a =
    let n = Array.length td.memoire in
    if td.curseur = n then
    begin
        let taille = max (2 * n) 1 in
        let nouvelle_memoire = Array.make taille a
    in
        for i = 0 to n-1 do
            nouvelle_memoire.(i) <- td.memoire.(i)
        done ;
        td.memoire <- nouvelle_memoire
    end ;
    td.memoire(td.curseur) <- a ;
    td.curseur <- td.curseur + 1
```

Programmation par contrat

Les postconditions, préconditions, variant et invariants permettent de définir un paradigme de programmation appelé *programmation par contrat* qui cherche à minimiser le nombres d'erreurs des programmes.

Exemple du DM

```
let vers_entier t =
    let rec aux k facteur acc = match k with
        |-1 -> acc
        | _ ->
            let terme =
                if t.(k) = 1
                    then facteur
                    else 0
            in aux (k-1) (facteur * 2) (acc + terme)
    in
    aux (taille_entier - 1) 1 0
```

Exemple du DM

```
let vers_entier t =
    let rec aux k facteur acc = match k with
        |-1 -> acc
        | _ ->
            let terme =
                if t.(k) = 1
                    then facteur
                    else 0
            in aux (k-1) (facteur * 2) (acc + terme)
    in
    aux (taille_entier - 1) 1 0
```

Comment montrer la terminaison ? La correction ?

Fonction 91 de McCarthy (1)

On définit la fonction f de la manière suivante :

```
let rec f n = match n >100 with
| true -> n-10
| false -> f ( f (n+ 11))
```

Combien vaut cette fonction pour les valeurs de n inférieurs ou égale à 101 ?

Fonction 91 de McCarthy (1)

On définit la fonction f de la manière suivante :

```
let rec f n = match n >100 with
| true -> n-10
| false -> f ( f (n+ 11))
```

Combien vaut cette fonction pour les valeurs de n inférieurs ou égale à 101 ?
Après quelques essais, on peut conjecturer que, pour tout $n \leqslant 101$, le calcul termine et $f(n) = 91$.

Fonction 91 de McCarthy (1)

On définit la fonction f de la manière suivante :

```
let rec f n = match n >100 with
| true -> n-10
| false -> f ( f (n+ 11))
```

Combien vaut cette fonction pour les valeurs de n inférieurs ou égale à 101 ?
Après quelques essais, on peut conjecturer que, pour tout $n \leqslant 101$, le calcul termine et $f(n) = 91$.
Il y a un nombre fini de valeurs, on peut le prouver en faisant une vérification exhaustive :

```
    let rec verifier n = match n>100 with
    | true -> true
    | false -> (f n = 91) && (verifier (n+1))
```

Fonction 91 de McCarthy (2)

```
let rec f n = match n >100 with
| true -> n-10
| false -> f ( f (n+ 11))
```

Fonction 91 de McCarthy (2)

```
1 let rec f n = match n >100 with
2 | true -> n-10
3 | false -> f ( f (n+ 11))
```

On peut aussi faire une démonstration :

Fonction 91 de McCarthy (2)

```
let rec f n = match n >100 with
| true -> n-10
| false -> f ( f (n+ 11))
```

On peut aussi faire une démonstration :

On remarque que pour $90 \leqslant n \leqslant 100$, on a $f(n) = f(f(n+11)) = f(n+1)$. On en déduit que $f(n) = f(101) = 91$ pour tout $90 \leqslant n \leqslant 101$.

Fonction 91 de McCarthy (2)

```
1 let rec f n = match n >100 with
2 | true -> n-10
3 | false -> f ( f (n+ 11))
```

On peut aussi faire une démonstration :
On remarque que pour $90 \leqslant n \leqslant 100$, on a
$f(n) = f(f(n+11)) = f(n+1)$. On en déduit que
$f(n) = f(101) = 91$ pour tout $90 \leqslant n \leqslant 101$.
Ensuite, on remarque que pour tout $79 \leqslant n \leqslant 89$, on a
$f(n) = f(f(n+11)) = f(91) = 91$.

Fonction 91 de McCarthy (2)

```
let rec f n = match n >100 with
| true -> n-10
| false -> f ( f (n+ 11))
```

On peut aussi faire une démonstration :
On remarque que pour $90 \leqslant n \leqslant 100$, on a
$f(n) = f(f(n+11)) = f(n+1)$. On en déduit que
$f(n) = f(101) = 91$ pour tout $90 \leqslant n \leqslant 101$.
Ensuite, on remarque que pour tout $79 \leqslant n \leqslant 89$, on a
$f(n) = f(f(n+11)) = f(91) = 91$.
Par itération, on montre le résultat attendu.

Démonstration par ordinateur

Théorème 2 : Théorème des 4 Couleurs

On peut colorier toute carte planaire en utilisant au plus 4 couleurs sans que deux pays limitrophes aient la même couleur.

Démonstration par ordinateur

Théorème 2 : Théorème des 4 Couleurs

On peut colorier toute carte planaire en utilisant au plus 4 couleurs sans que deux pays limitrophes aient la même couleur.

En 76, Kenneth Appel et Wolfang Haken prétendent avoir démontré le théorème des 4 couleurs grâce à une étude exhaustive des différents cas possibles. Le problème est qu'il est nécessaire de démontrer la validation du programme.

Démonstration par ordinateur

Théorème 2 : Théorème des 4 Couleurs

On peut colorier toute carte planaire en utilisant au plus 4 couleurs sans que deux pays limitrophes aient la même couleur.

En 76, Kenneth Appel et Wolfang Haken prétendent avoir démontré le théorème des 4 couleurs grâce à une étude exhaustive des différents cas possibles. Le problème est qu'il est nécessaire de démontrer la validation du programme.

En 2005, le théorème des 4 couleurs est montré dans un assistant de preuve, un outil de démonstration assisté par ordinateur, mais il n'y a toujours pas de démonstration qui peut s'en passer.

Dichotomie (1)

Il y a une erreur traître dans cette implémentation :

```
1  let recherche t x =
2  let n = Array.length t in
3  let rec aux i j =
4      if i>j then false
5      else let k = (i+j)/2 in
6          if x = t.(k) then true
7          else
8          if x> t.(k) then aux (k+1) j
9          else aux i (k-1)
10 in aux 0 (n-1)
```

Dichotomie (1)

Il y a une erreur traître dans cette implémentation :

```
let recherche t x =
let n = Array.length t in
let rec aux i j =
    if i>j then false
    else let k = (i+j)/2 in
        if x = t.(k) then true
        else
        if x> t.(k) then aux (k+1) j
        else aux i (k-1)
in aux 0 (n-1)
```

Le calcul de k par $\frac{i+j}{2}$ peut entraîner un dépassement de mémoire si i et j sont déjà grands.

Dichotomie (2)

```
let recherche t x =
    let n = Array.length t in
    let rec aux i j =
        if i>j then false
        else let k = i + (j-i)/2 in
            if x = t.(k) then true
            else
            if x> t.(k) then aux (k+1) j
            else aux i (k-1)
    in aux 0 (n-1)
```

Dichotomie (2)

```
let recherche t x =
    let n = Array.length t in
    let rec aux i j =
        if i>j then false
        else let k = i + (j-i)/2 in
            if x = t.(k) then true
            else
            if x> t.(k) then aux (k+1) j
            else aux i (k-1)
    in aux 0 (n-1)
```

De cette manière, on peut esquiver le cas du dépassement lors du calcul de $i + j$.

Dichotomie (2)

```
let recherche t x =
    let n = Array.length t in
    let rec aux i j =
        if i>j then false
        else let k = i + (j-i)/2 in
            if x = t.(k) then true
            else
            if x> t.(k) then aux (k+1) j
            else aux i (k-1)
    in aux 0 (n-1)
```

De cette manière, on peut esquiver le cas du dépassement lors du calcul de $i + j$.
En pratique, le problème ne survient pas si on encode les entiers sur 64 bits.

Fonction avec récursivité croisée (1)

$$F(0) = 1$$
$$M(0) = 0$$
$$\forall n > 0 \; F(n) = n - M(F(n-1))$$
$$\forall n > 0 \; M(n) = n - F(M(n-1))$$

Fonction avec récursivité croisée (1)

$$F(0) = 1$$
$$M(0) = 0$$
$$\forall n > 0 \; F(n) = n - M(F(n-1))$$
$$\forall n > 0 \; M(n) = n - F(M(n-1))$$

Comment implémenter ces fonctions ?

Fonction avec récursivité croisée (1)

$$F(0) = 1$$
$$M(0) = 0$$
$$\forall n > 0\ F(n) = n - M(F(n-1))$$
$$\forall n > 0\ M(n) = n - F(M(n-1))$$

Comment implémenter ces fonctions ?

On utilise le mot-clef `and` :

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Fonction avec récursivité croisée (1)

$$F(0) = 1$$
$$M(0) = 0$$
$$\forall n > 0\ F(n) = n - M(F(n-1))$$
$$\forall n > 0\ M(n) = n - F(M(n-1))$$

Comment implémenter ces fonctions ?

On utilise le mot-clef `and` :

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Comment en montrer la terminaison ?

Fonction avec récursivité croisée (2)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Fonction avec récursivité croisée (2)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Il suffit de montrer que M et F sont bien définies. On peut montrer que lors du calcul de $F(n)$ on doit calculer $M(n_0)$ pour $n_0 \leqslant n$, et pour le calcul de $M(n)$, on doit calculer $F(n_0)$ pour $n_0 < n$.

Fonction avec récursivité croisée (2)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Il suffit de montrer que M et F sont bien définies. On peut montrer que lors du calcul de $F(n)$ on doit calculer $M(n_0)$ pour $n_0 \leqslant n$, et pour le calcul de $M(n)$, on doit calculer $F(n_0)$ pour $n_0 < n$.
On peut montrer que $0 \leqslant F(n+1) - F(n) \leqslant 1$ et $0 \leqslant M(n+1) - M(n) \leqslant 1$ pour tout n. La démonstration par récurrence forte de cette propriété nous suffit à montrer que F et M sont bien définies.

Fonction avec récursivité croisée (3)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Fonction avec récursivité croisée (3)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Pour l'initialisation, on remarque que $M(0) = 0$, $M(1) = 0$, $M(2) = 1$, $F(0) = 1$, $F(1) = 1$ et $F(2) = 2$.

Fonction avec récursivité croisée (3)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Pour l'initialisation, on remarque que $M(0) = 0$, $M(1) = 0$, $M(2) = 1$, $F(0) = 1$, $F(1) = 1$ et $F(2) = 2$.
Pour l'hérédité :

$$\begin{aligned} F(n+1) - F(n) &= n + 1 - M(F(n)) - (n - M(F(n-1))) \\ &= 1 - (M(F(n)) - M(F(n-1))) \end{aligned}$$

Fonction avec récursivité croisée (3)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Pour l'initialisation, on remarque que $M(0) = 0$, $M(1) = 0$, $M(2) = 1$, $F(0) = 1$, $F(1) = 1$ et $F(2) = 2$.
Pour l'hérédité :

$$\begin{aligned} F(n+1) - F(n) &= n + 1 - M(F(n)) - (n - M(F(n-1))) \\ &= 1 - (M(F(n)) - M(F(n-1))) \end{aligned}$$

Or, par hypothèse de récurrence, $F(n) = F(n-1)$ ou $F(n) = F(n-1) + 1$, et donc $M(F(n)) - M(F(n-1)) = 0$, ou $M(F(n)) - M(F(n-1)) = 1$. (En effet, $F(n) \leqslant n$ pour $n > 0$)

Fonction avec récursivité croisée (3)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Pour l'initialisation, on remarque que $M(0) = 0$, $M(1) = 0$, $M(2) = 1$, $F(0) = 1$, $F(1) = 1$ et $F(2) = 2$.
Pour l'hérédité :

$$\begin{aligned} F(n+1) - F(n) &= n + 1 - M(F(n)) - (n - M(F(n-1))) \\ &= 1 - (M(F(n)) - M(F(n-1))) \end{aligned}$$

Or, par hypothèse de récurrence, $F(n) = F(n-1)$ ou $F(n) = F(n-1) + 1$, et donc $M(F(n)) - M(F(n-1)) = 0$, ou $M(F(n)) - M(F(n-1)) = 1$. (En effet, $F(n) \leqslant n$ pour $n > 0$)
Donc F et M sont bien définies et notre programme termine pour toute valeur de $n \geqslant 0$.

Fonction avec récursivité croisée (4)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Fonction avec récursivité croisée (4)

```
1 let rec f n = match n with
2 | 0 -> 1
3 | _ -> n - m (f (n-1))
4 and m n = match n with
5 | 0 -> 0
6 | _ -> n - f (m (n-1))
```

Pour l'hérédité :

$$\begin{aligned} F(n+1) - F(n) &= n + 1 - M(F(n)) - (n - M(F(n-1))) \\ &= 1 - (M(F(n)) - M(F(n-1))) \end{aligned}$$

Fonction avec récursivité croisée (4)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Pour l'hérédité :

$$\begin{aligned} F(n+1) - F(n) &= n + 1 - M(F(n)) - (n - M(F(n-1))) \\ &= 1 - (M(F(n)) - M(F(n-1))) \end{aligned}$$

Or, par hypothèse de récurrence, $F(n) = F(n-1)$ ou $F(n) = F(n-1) + 1$, et donc $M(F(n)) - M(F(n-1)) = 0$, ou $M(F(n)) - M(F(n-1)) = 1$. (En effet, $F(n) \leqslant n$ pour $n > 0$)

Fonction avec récursivité croisée (4)

```
let rec f n = match n with
| 0 -> 1
| _ -> n - m (f (n-1))
and m n = match n with
| 0 -> 0
| _ -> n - f (m (n-1))
```

Pour l'hérédité :

$$\begin{aligned} F(n+1) - F(n) &= n + 1 - M(F(n)) - (n - M(F(n-1))) \\ &= 1 - (M(F(n)) - M(F(n-1))) \end{aligned}$$

Or, par hypothèse de récurrence, $F(n) = F(n-1)$ ou $F(n) = F(n-1) + 1$, et donc $M(F(n)) - M(F(n-1)) = 0$, ou $M(F(n)) - M(F(n-1)) = 1$. (En effet, $F(n) \leqslant n$ pour $n > 0$)
Donc F et M sont bien définies et notre programme termine pour toute valeur de $n \geqslant 0$.