Calculatrices interdites ; **Encadrer les solutions** ; Séparer les questions d'un trait

Exercice

On étudie deux fonctions permettant de numéroter les éléments de $\mathbb{N}^2$.

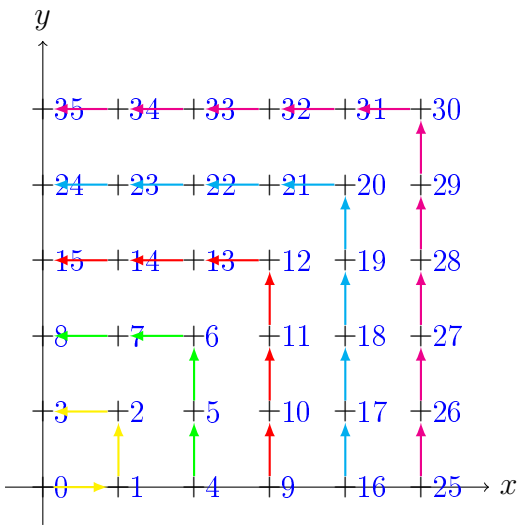
1. On considère la fonction `carre` définie par :

$$\forall (x,y) \in \mathbb{N}^2 : \texttt{carre}(x,y) = \max^2(x,y) + \max(x,y) + y - x$$

(a) La fonction `carre` est de type : `int -> int -> int.`

```
(* val carre : int -> int -> int = <fun> *)
let carre x y =
 let m = max x y in
  m * m + m + y - x
;;
```

(b) La fonction `carre` permet d'effectuer une numérotation dite en carrée des éléments de $\mathbb{N}^2$. Justifier par un graphique cette terminologie.
Numérotation en carré des couples de $\mathbb{N}^2$:



(c) La fonction `carre_rec` est définie sur $\mathbb{N} \times \mathbb{N}$ par les relations [$\mathfrak{RC}$] :
$\forall (m,n) \in \mathbb{N}^2$:

$$\begin{cases} \texttt{carre\_rec}(m,0) &= & m^2 & \text{lorsque} & n = 0 \\ \texttt{carre\_rec}(m,n) &= & \texttt{carre\_rec}(m,n-1)+1 & \text{lorsque} & 1 \leqslant n \leqslant m \\ \texttt{carre\_rec}(m,n) &= & \texttt{carre\_rec}(m+1,n)+1 & \text{lorsque} & n > m \end{cases}$$

La fonction `carre_rec` est de type : `int -> int -> int.`

```
(* val carre_rec : int -> int -> int = <fun>  *)
let rec carre_rec x y =
  match x, y with
   | m, 0 -> m * m
   | m, n when n <= m -> carre_rec m (n - 1) + 1
   | m, n when n > m -> carre_rec (m + 1) n + 1
;;
```

(d) Notons P_n l'assertion : " `carre_rec`(m,n) termine pour tout $m \in \mathbb{N}$."
- Avec le premier cas, `carre_rec`$(m,0)$ termine.
  L'assertion P_0 est vérifiée.
- Supposons P_n pour $n \in \mathbb{N}$ fixé, ie que `carre_rec`(m,n) termine pour tout $m \in \mathbb{N}$.

- Soit $m \in \mathbb{N}$.
  $\diamond$ Si $m > n$ ie $m \geqslant n+1$:
  Avec P_n, `carre_rec`(m,n) termine.
  Or, avec le deuxième cas, comme $n+1 \leqslant m$:
  `carre_rec`$(m, n+1) =$ `carre_rec`$(m,n) + 1$
  Donc `carre_rec`$(m, n+1)$ termine.
  Remarquons aussi que `carre_rec`$(n+1, n+1)$ termine.
  $\diamond$ Si $m \leqslant n$:
  Notons Q_k l'assertion "`carre_rec`$(n+1-k, n+1)$ termine pour tout $k \in [|0, n+1|]$."
  $*$Avec la remarque précédente, on a Q_0.
  $*$Supposons Q_k pour $k \in [|0, m|]$ fixé.
  $*$`carre_rec`$(n+1-k, n+1)$ termine
  Or, avec le troisième cas, comme $n+1 > n+1-(k+1)$:
  `carre_rec`$(n+1-(k+1), n+1) =$ `carre_rec`$(n+1-k, n+1) + 1$
  Donc `carre_rec`$(n+1-(k+1), n+1)$ termine
  D'où l'assertion Q_{k+1} est vérifiée.
  Par récurrence finie : $\boxed{\forall k \in [|0, n+1|] : Q_k \text{ est vérifiée.}}$
  Donc l'assertion P_{n+1} est vérifiée.
  Par récurrence : $\boxed{\forall n \in \mathbb{N}, \text{ l'assertion } P_n \text{ est vérifiée.}}$

  Ainsi, pour tout $(n,m) \in \mathbb{N}^2$, `carre_rec`(n,m) termine.

  Remarquons que cette récurrence permet également de montrer que `carre_rec` est bien défine, ie que, pour tout $(n,m) \in \mathbb{N}^2$, (n,m) possède une unique image par `carre_rec`. En efffet, il suffit de remplacer P_n par l'assertion : "`carre_rec`(m,n) termine et possède une unique valeur pour tout $m \in \mathbb{N}$."

(e) Notons C la fonction `carre`. Il suffit de monter que C vérifie les relations $[\mathfrak{RC}]$
Soit $(m,n) \in \mathbb{N}^2$.
- $C(m,0) = m^2 + m + 0 - m = m^2$.
- Si $n \leqslant m$ alors $n-1 \leqslant m$ et :
  $C(m,n) = m^2 + m + n - m = m^2 + n$.
  $C(m, n-1) = m^2 + m + n - 1 - m = m^2 + n - 1$.
  Donc $C(m,n) = C(m, n-1) + 1$
- Si $n > m$ ie $n \geqslant m+1$:
  $C(m,n) = n^2 + n + n - m = n^2 + 2n - m$
  $C(m+1, n) = n^2 + n + n - (m+1) = n^2 + 2n - m - 1$
  Donc $C(m,n) = C(m+1, n) + 1$.

Ainsi C vérifie les relations $[\mathfrak{RC}]$.
La fonction `carre_rec` est donc égale à la fonction `carre`.

La correction de la fonction `carre_rec` est vérifiée.

2. On considère la fonction `triangle` définie par :

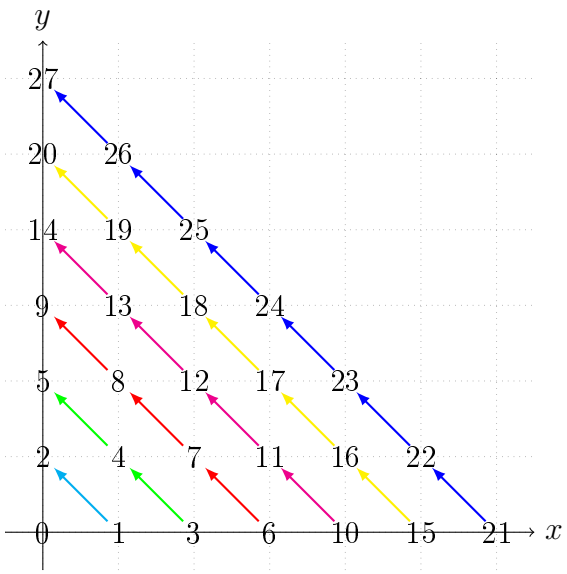
$$\forall (x,y) \in \mathbb{N}^2 : \texttt{triangle}(x,y) = \frac{(x+y)(x+y+1)}{2} + y$$

Dans cette question, il est inutile de préciser les types des fonctions considérées.

(a) En CAML , la fonction `triangle` est :

```
(* val triangle : int -> int -> int = <fun>  *)
let triangle x y =
  let s = x + y in
    s * (s + 1) / 2 + y
;;
```

(b) La fonction `triangle` permet d'effectuer une numerotation dite en triangle des éléments de $\mathbb{N}^2$. Justifier par un graphique cette terminologie.
Numérotation en triangle des couples de $\mathbb{N}^2$:



(c) Définir récursivement sur $\mathbb{N} \times \mathbb{N}$ une fonction `triangle_rec` coïncidant avec la fonction `triangle`.
La fonction `triangle_rec` est définie sur $\mathbb{N} \times \mathbb{N}$ par $\forall m \in \mathbb{N}$:

$$\begin{cases} \texttt{triangle\_rec}(m,0) &= \frac{m(m+1)}{2} & \text{lorsque} \quad n = 00 \\ \texttt{triangle\_rec}(m,n) &= \texttt{triangle\_rec}(m+1,n-1)+1 & \text{lorsque} \quad n \geqslant 1 \end{cases}$$

(d) Définir en CAML la fonction `triangle_rec`.

```
(* val triangle_rec : int -> int -> int = <fun> *)
let rec triangle_rec x y =
  match x, y with
  | _, 0 -> x * ( x + 1) / 2
  | _, _ -> triangle_rec (x + 1) (y - 1) + 1
;;
```

(e) Prouver la terminaison de la fonction `triangle_rec`.
Notons T_n l'assertion `triangle_rec`(m,n) termine pour tout $m \in \mathbb{N}$.

- `triangle_rec`$(m,0)$ termine avec le premir cas.
- Supposons, pour $n \in \mathbb{N}$ fixé, que l'assertion T_n est vérifiée ie que `triangle_rec`(m,n) termine.
- Avec l'assertion T_n, `triangle_rec`$(m+1,n)$
  Or `triangle_rec`$(m,n+1)$ = `triangle_rec`$(m+1,n)$ +1 donc `triangle_rec`$(m,n+1)$ termine.
  d'où l'assertion T_{n+1} est vérifiée

Par récurrence, pour tout $n \in \mathbb{N}$, l'assertion T_n est vérifiée.

Ainsi, pour tout $(m,n) \in \mathbb{N}^2$, `triangle_rec`(m,n) termine.

Exercice

1. On définit le type `signe` avec.

```
type signe =
  | Plus
  | Moins
;;
```

(a) Définir en CAML la fonction `changement_signe` de prenant en entrée un signe *signe*1 et retournant le signe opposé à *signe*1. Ecrire au préalable son type.

```
(* changement_signe : signe -> signe = <fun> *)
let changement_signe signe =
 if signe = Plus then Moins else Plus
;;
```

(b) Définir en CAML la fonction `regle_des_signes` de prenant en entrée deux signes, *signe*1 et *signe*2 et retournant le signe obtenu en appliquant la règle des signes à *signe*1 et *signe*2. Ecrire au préalable son type.

```
(* regle_des_signes : 'a -> 'a -> signe = <fun> *)
let regle_des_signes signe1 signe2 =
  if signe1 = signe2 then  Plus else Moins
;;
```

2. Après avoir défini une type `limite` comportant quatre constructeurs dont 3 paramétrés, on peut écrire la fonction `oppose`, notée ci dessous, et de type `limite ->limite`, qui retourne la valeur de la limite de l'opposée d'une suite réelle de limite *limite*1.

```
let oppose limite =
  match limite with
  | FormeIndeterminee -> FormeIndeterminee
  | Infini signe -> Infini (changement_signe signe)
  | Zero signe -> Zero (changement_signe signe)
  | Nombre valeur -> Nombre (-. valeur)
;;
```

(a) Définir le type `limite`

```
type limite =
   | Infini of signe
   | Zero of signe
   | Nombre of float
   | FormeIndeterminee
;;
```

(b) Que vaut `oppose (Infini Plus);;` ?

```
oppose (Infini Plus);;
- : limite = Infini Moins
```

(c) Que vaut `oppose (Nombre 48);;` ?

```
oppose (Nombre 48);;
(* Cette expression est de type int,
mais est utilisée avec le type float.*)
(* par contre: *)
oppose (Nombre 48.);;
- : limite = Nombre -48.0
```

3. Définir en CAML la fonction `inverse` prenant en entrée une limite notée *limite*1 et retournant la limite de l'inverse d'une suite de limite *limite*1. Ecrire au préalable son type.

```
(* inverse : limite -> limite = <fun> *)
let inverse limite =
  match limite with
  | FormeIndeterminee -> FormeIndeterminee
  | Infini signe -> Zero signe
  | Zero signe -> Infini signe
  | Nombre valeur when valeur = 0. -> FormeIndeterminee
  | Nombre valeur  ->  Nombre ( 1. /. valeur)
;;
```

4.(a) Définir en CAML la fonction `somme` prenant en entrée deux limites notées *limite*1 et *limite*2 et retournant la limite de la somme de deux suites de limites respectives *limite*1 et *limite*2. Ecrire au préalable son type.
On devra obtenir :

```
somme (Zero Moins) (Zero Plus) ;;
(* - : limite = Nombre 0.0 *)
somme (Zero Plus) (Zero Plus) ;;
(* - : limite = Zero Plus *)
```

```
(* somme : limite -> limite -> limite = <fun> *)
let somme  limite1 limite2 =
  match limite1, limite2 with
  (* Avec une forme indeterminee *)
  | FormeIndeterminee, _ -> FormeIndeterminee
  | _, FormeIndeterminee -> FormeIndeterminee
  (* Avec un infini *)
  | Infini signe1, Infini signe2 when signe1 <> signe2 -> FormeIndeterminee
  | Infini _, _ -> limite1
  | _, Infini _ -> limite2
  (* Avec une limite valant Zero *)
  | Nombre _, Zero _ -> limite1
  | Zero _,  Nombre _ -> limite2
  | Zero signe1, Zero signe2 when signe1 = signe2 -> limite1
  | Zero _, Zero _ -> Nombre 0.
  (* Avec deux limites reelles *)
  | Nombre x1, Nombre x2 -> Nombre ( x1 +. x2)
;;
```

(b) Définir en CAML la fonction `difference` prenant en entrée deux limites notées *limite*1 et *limite*2 et retournant la limite de la différence de deux suites de limites respectives *limite*1 et *limite*2. Ecrire au préalable son type.

```
(* difference : limite -> limite -> limite = <fun> *)
let difference limite1 limite2 =
  somme  limite1 (oppose limite2)
;;
```

5. On souhaite obtenir la fonction `produit` prenant en entrée deux limites notées *limite*1 et *limite*2 et retournant la limite de la somme de deux suites de limites respectives *limite*1 et *limite*2. On ne demande pas le type de cette fonction.
Recopier et compléter.
On utilisera impérativement deux couleurs, l'une pour le script donné et l'autre pour la partie à compléter.

```
let produit  limite1 limite2 =
  match limite1, limite2 with
  (* Avec une forme indeterminee *)
  | FormeIndeterminee, _ -> FormeIndeterminee
  | _, FormeIndeterminee -> FormeIndeterminee
  (* infini multiplie par infini *)
  | Infini signe1, Infini signe2 -> Infini (regle_des_signes signe1 signe2)
  (* infini multiplie par 0 *)
  | Infini _, Zero _ -> FormeIndeterminee
  | Zero _, Infini _ -> FormeIndeterminee
  | Infini _, Nombre 0. -> FormeIndeterminee
  | Nombre 0., Infini _ -> FormeIndeterminee
  (* infini multiplie par un reel non nul *)
  | Infini signe1, Nombre valeur2   ->
    let signe2 = if valeur2 > 0. then Plus else Moins in
          Infini (regle_des_signes signe1 signe2)
  | Nombre valeur1, Infini signe2  ->
        let signe1 = if valeur1 > 0. then Plus else Moins in
          Infini (regle_des_signes signe1 signe2)
  (* limite nulle, mais on precise le signe si possible *)
  | Zero signe1, Zero signe2 -> Zero (regle_des_signes signe1 signe2)
  | Zero _, Nombre 0. -> Nombre 0.
  | Nombre 0., Zero_ -> Nombre 0.
  | Zero signe1, Nombre valeur2 ->
        let signe2 = if valeur2 > 0. then Plus else Moins in
```

```
                Zero (regle_des_signes signe1 signe2)
| Nombre valeur1, Zero signe2 -> (* completer *)
        let signe1 = if valeur1 > 0. then Plus else Moins in
          Zero (regle_des_signes signe1 signe2)
(* Avec deux limites reelles *)
| Nombre valeur1, Nombre valeur2 -> Nombre (valeur1 *. valeur2)
      ;;
```

Fin du sujet.