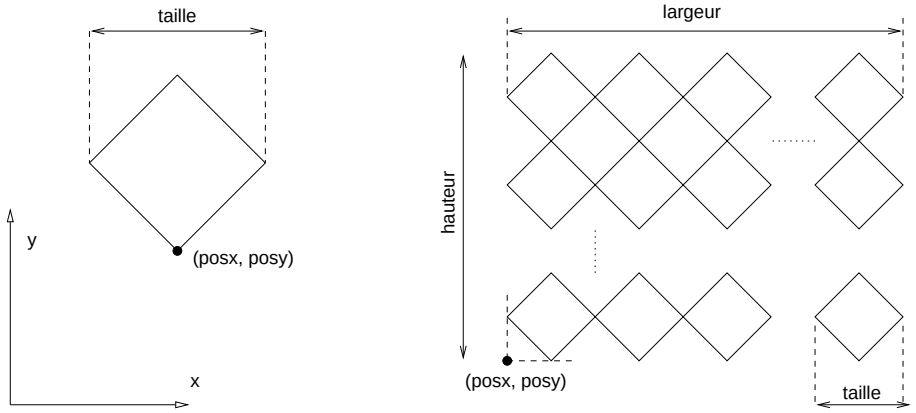
CS110 – Devoir surveillé – Correction

Note : il y a toujours plusieurs façons correctes de résoudre un problème algorithmique. Les solutions proposées ci-dessous ne sont donc pas les seules possibles.

Procédures de dessin

On dispose d'un table traçante que l'on peut commander avec les trois procédures suivantes :
– `procédure position(x,y:réel)` déplace le crayon (sans écrire) à la position absolue (x,y).
– `procédure déplace(dx,dy:réel)` déplace le crayon (sans écrire) de dx unités sur l'axe des x et dy unités sur l'axe des y.
– `procédure trace(dx,dy:réel)` idem procédure `déplace` mais trace un trait pendant le déplacement.
On supposera que la table est de taille infinie.



1. (**3 pts**) Écrivez une procédure qui dessine un carré "sur la pointe", d'une taille donnée et dont le coin inférieur est à une position donnée, comme illustré dans la partie gauche de la figure ci-dessus (cette procédure prend donc trois paramètres).

   .....

   Il faut bien comprendre à quoi correspondent les trois procédures de commande de la table traçante. `position` place le crayon à une valeur absolue, alors que `déplace` et `trace` partent de la position courante, et effectuent un déplacement de `dx` et `dy` unités. Noter que tous les paramètres sont réels.

   Les trois paramètres de la procédure demandée sont l'abscisse et l'ordonnée du coin inférieur du carré, et sa taille.

   Il s'agit de données, ce seront donc des paramètres d'entrée, passés par valeur. Les noms à utiliser pour les paramètres formels sont libres, mais la figure du carré suggérait les noms `posx`, `posy` et `taille`.

```
procédure dessine_carré(posx, posy:réel; taille:réel)
(* dessine un carré ''sur la pointe'' en (posx, posy) de taille *)
début
    position(posx, posy)
    trace(-taille/2, taille/2)
    trace(taille/2, taille/2)
    trace(taille/2, -taille/2)
    trace(-taille/2, -taille/2)
fin
```

2. (**3 pts**) Écrivez une procédure qui utilise la procédure précédente pour dessiner le même carré avec en plus ses diagonales.

   .....

   Le problème est très proche, ses données sont les mêmes, on trouve donc les trois mêmes paramètres. Puisqu'on vient de définir `dessine_carré`, on peut maintenant l'utiliser. On remarque qu'elle laisse le crayon à la position du coin inférieur. Cela dit, la spécification de `dessine_carré` ne le précise pas. Il est donc plus prudent de replacer le crayon.

```
procédure carré_diag(posx, posy:réel; taille:réel)
(* dessine un carré avec ses diagonales *)
début
     dessine_carré(posx, posy, taille)
     position(posx, posy)
     trace(0, taille)
     déplace(-taille/2, -taille/2)
     trace(taille, 0)
fin
```

3. (**6 pts**) Écrivez une procédure qui dessine une mosaïque de la forme décrite partie droite de la figure ci-dessus. Elle prend en paramètres la taille des carrés, la position de la mosaïque, ses largeur et hauteur en nombre de carrés. Si cela vous parait trop compliqué, vous pouvez commencer par ne dessiner qu'une ligne de la mosaïque.

   .....

   Tout d'abord, quelles sont les données du problème ? Elles sont données explicitement dans l'énoncé :

   (a) La taille des carrés.

   (b) Les coordonnées de la mosaïque. Le schéma montre que l'abcisse (`posx`) des carrés sur la première colonne est égale à l'abcisse de la mosaïque plus `taille/2`.

   (c) Les largeur et hauteur de la mosaïque. L'énoncé spécifie qu'elles sont exprimées en **nombre de carrés**, ce seront donc des entiers.

   On a déjà l'en-tête de la procédure :

```
procédure mosaique(posx, posy:réel; larg, haut:entier; taille:réel)
(* dessine une mosaique de larg×haut carrés *)
(* de taille taille en (posx,posy)          *)
```

   Comment la procédure va-t-elle... procéder ? On doit dessiner `larg` carrés en largeur et `haut` carrés en hauteur. Nous pouvons numéroter les colonnes de 0 à `larg-1` et les lignes de 0 à `haut-1`. Il y a deux niveaux de répétitions, donc deux boucles imbriquées. Le nombre d'itérations pour chacune est connu, ce seront donc des boucles `pour`. Pour les compteurs de boucle, choisissons des noms qui évitent de s'embrouiller : `l` pour la boucle en largeur, `h` pour celle en hauteur.

```
procédure mosaique(posx, posy:réel; larg, haut:entier; taille:réel)
(* dessine une mosaique de larg×haut carrés *)
(* de taille taille en (posx,posy)          *)
var l,h:entier
début
     pour l de 0 à larg-1 faire
         pour h de 0 à haut-1 faire
              ...
         fin pour
     fin pour
fin
```

   Enfin, on va utiliser la procédure `dessine_carré` que l'on a défini précédemment. Celle-ci permet de dessiner un carré à une position donnée. Il nous reste à déterminer la position du carré qui est à la colonne `l` et à la ligne `h`, ce qui ne pose pas de problème particulier si l'on regarde bien le schéma.

```
procédure mosaique(posx, posy:réel; larg, haut:entier; taille:réel)
(* dessine une mosaique de larg×haut carrés *)
(* de taille taille en (posx,posy)          *)
var l,h:entier
début
     pour l de 0 à larg-1 faire
         pour h de 0 à haut-1 faire
             dessine_carré(posx+l×taille+taille/2, posy+h×taille, taille)
         fin pour
```

```
        fin pour
    fin
```

Beaucoup d'entre vous ont essayé de faire des déplacements du crayon à chaque itération afin de le placer à la bonne position pour le prochain carré. C'est possible mais il fallait alors définir une autre procédure pour dessiner un carré, par exemple `dessine_carré_courant`. Celle-ci ne prend que la taille en paramètre et dessine le carré à la position courante, au lieu de le dessiner à la position passée en paramètres...

Tableaux

Soient les déclarations suivantes :

```
const taille=100
type  tab=tableau[1..taille] d'entiers
```

1. (**1 pt**) Écrivez une fonction qui prend un entier en paramètre et retourne sa valeur absolue.

   .....

   Erreur à ne pas faire : faire deux `si/alors/finsi` consécutifs avec chacun une instruction `retour`. Voir la fiche de référence page 7, note 2. `retour` doit être la dernière instruction à exécuter dans la fonction.

   Il est inutile de passer par une variable intermédiaire à laquelle on affecte l'oppsé de `n`, ou de faire des multiplications par -1 (la fiche de référence mentionne que `-` est aussi l'opérateur de la négation). En clair si *`exp`* est une expression numérique, `-`*`exp`* est son opposée.

   ```
   fonction abs(n:entier):entier
   (* retourne valeur absolue de n *)
   début
       si n>0 alors
                retour(n)
       sinon
                retour(-n)
       fin si
   fin
   ```

2. On veut déterminer, pour une valeur de type `tab`, quel est le plus grand élément en valeur absolue.

   (a) (**1 pt**) Donnez l'en-tête d'une fonction qui pourrait résoudre ce problème. Justifiez le mode de passage du ou des paramètres et le type de la valeur retournée.

   .....

   On a un paramètre d'entrée (c'est une donnée) de type `tab`. On le passe donc par valeur (sans le mot-clé `var`).

   Ici, je dois avouer qu'il y a plusieurs interprétations possibles pour l'énoncé. Qu'entend-on par "déterminer quel est l'élément le plus grand en valeur absolue" ? Doit-on retourner :

   i. la valeur absolue du plus grand élément en valeur absolue,

   ii. la valeur (tout court) du plus grand élément en valeur absolue,

   iii. l'indice dans le tableau du plus grand élément en valeur absolue ?

   La plupart d'entre vous ont opté pour le choix i, ce qui est ce que j'avais en tête. Nous ferons de même dans ce corrigé. Mais les deux autres choix sont également tout à fait acceptables (et sans doute même plus que le premier...)

   Quoi qu'il en soit, dans tous les cas la valeur de retour sera un entier. Certains d'entre vous ont choisi d'appeler le paramètre `tab`. Très mauvaise idée ! Ce nom est déjà utilisé pour le type.

   ```
   fonction max_en_abs(t:tab):entier
   (* retourne le plus grand élément de t en valeur absolue *)
   ```

(b) (**3 pts**) Rédigez la fonction en utilisant une boucle `pour`.

.....

Il s'agit d'un cas classique de "parcours avec accumulation". Nous cherchons la valeur absolue de l'élément le plus grand en valeur absolue. Appelons `max` cette valeur. Au départ, faute de mieux, on lui donne la valeur absolue du premier élément du tableau.

Puis l'on compare `max` avec la valeur absolue de tous les autres éléments du tableau. Chaque fois que l'on trouve une valeur plus grande, on la copie dans `max`.

Ainsi, `max` contient à tout moment la plus grande valeur absolue des éléments du tableau que l'on a parcourus. À la fin de la boucle, on a parcouru tous les éléments, `max` contient donc bien ce que l'on cherche

Variantes possibles : on pouvait initiliser `max` à une valeur plus petite ou égale à toutes les valeurs absolues que l'on peut trouver dans le tableau (par exemple 0 ou -1) et la comparer à tous les éléments.

Par contre, c'est une erreur de commencer avec `max` égal à 1. Si le tableau ne contient que des 0, `max` restera égal à 1...

Noter aussi qu'il n'est absolument pas nécessaire de recopier toutes les valeurs absolues dans un deuxième tableau pour faire la recherche dans ce tableau.

```
fonction max_en_abs(t:tab):entier
(* retourne le plus grand élément de t en valeur absolue *)
var i, max:entier
début
    max ← abs(t[1])
    pour i de 2 à taille faire
         si abs(t[i])>max alors max ← abs(t[i]) fin si
    fin pour
    retour(max)
fin
```

(c) (**3 pts**) Rédigez la fonction en utilisant une boucle `tant que`.

.....

Pour remplacer une boucle `pour` par une `tant que`, il faut manipuler explicitement la variable compteur de boucle (fixer sa valeur initiale et l'incrémenter à la fin de chaque itération) et écrire la condition de continuation. Erreur courante : utiliser `i < taille` comme condition. On ne teste alors pas le dernier élément, `t[taille]`...

```
fonction max_en_abs(t:tab):entier
(* retourne le plus grand élément de t en valeur absolue *)
var i, max:entier
début
    max ← abs(t[1])
    i ← 2
    tant que i ≤ taille faire
         si abs(t[i])>max alors max ← abs(t[i]) fin si
         i ← i + 1
    fin tant que
    retour(max)
fin
```