Projet de langage

Le projet de langage se fait par groupe de deux élèves.

Chaque groupe déposera au secrétariat de l'IUP un rapport accompagné d'une disquette au plus tard le lundi 28 juin au matin. Chaque jour de retard sera sanctionné d'un point.
La disquette contiendra le code source intégral (fichiers C, `flex` et `bison`) de l'interpréteur développé lors du projet ainsi que quelques fichiers d'exemples.
Le rapport devra discuter les choix d'implémentation (grammaires, structures de données et algorithmes) et documenter l'utilisation de l'interpréteur. Le contenu de la section 2 devra obligatoirement être implémenté par l'interpréteur, mais il n'est pas interdit d'y ajouter des améliorations de votre cru, ou de traiter des parties de la section 3.

Lors d'une soutenance orale ultérieure (date à déterminer), chaque groupe fera la démonstration sur ordinateur de l'interpréteur et répondra aux questions des examinateurs. Une note pour l'ensemble du projet (rapport, disquette et soutenance) sera donnée à chaque élève.

1 Introduction

Le but de ce projet est de programmer en C, avec l'aide de `flex` et `bison`, un générateur d'images procédurales.

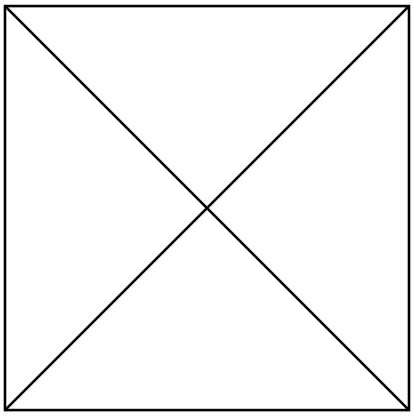
Une image procédurale est une image calculée à partir d'une fonction mathématique ou d'un programme informatique. On se limitera ici à des images en niveaux de gris.
Chaque programme est une suite d'instructions qui, à partir des coordonnées `X` et `Y` d'un point, se charge de calculer son intensité lumineuse `C`.

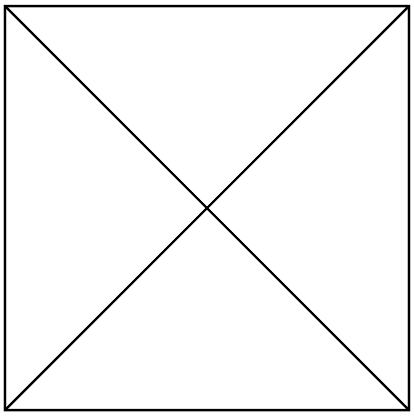
Exemple. Par exemple, les deux programmes :

```
/* Échiquier */
C = floor(X*10)+floor(Y*10) % 2 ;
```

```
/* Spirale */
XX = (X-0.5) * 30 ;
YY = (Y-0.5) * 30 ;
RR = (XX**2 + YY**2) ** 0.5 ;
X2 = XX*cos(RR) - YY*sin(RR) ;
if (X2>0 && X2<10) { C=0 ; } else { C=1 ; } ;
```

devront générer respectivement les images :





2 Description de l'interpréteur

L'interpréteur prend en argument un programme (dans un langage décrit à la section 2.1) et une taille d'image à générer, et écrit dans un fichier une image au format TGA (décrit à la section 2.3) de cette taille.

2.1 Le langage

Dans notre langage, les espaces, tabulations et retours à la ligne ne sont pas significatifs. De plus, tout texte mis entre les caractères `/*` et `*/` doit être considéré comme un commentaire et ignoré par l'interpréteur.

2.1.1 Les instructions

Un programme se compose d'une séquence d'instructions. Toutes les instructions se terminent par un point-virgule `;`.

Les affectations. L'instruction d'affectation s'écrit *variable*=*expression* `;`. Elle évalue l'expression donnée et met le résultat dans la variable précisée.
Un nom de variable est composé d'un nombre arbitraire de lettres et de chiffres, le premier caractère étant forcément une lettre (pour ne pas confondre avec le début d'un nombre). Contrairement au C, les variables n'ont pas besoin d'être déclarées avant d'être utilisées. De plus, toutes les variables et expressions sont de type flottant (`double` en C).
Les variables `X`, `Y`, et `C` ont une signification spéciale : `X` et `Y` correspond à la position courante et `C` à la couleur courante (voir section 2.2).

Les Tests. Un test s'écrit `if (`*expr*`) {` $inst_1 \cdots inst_n$ `};` ou `if (`*expr*`) {` $inst_1 \cdots inst_n$ `} else {` $inst'_1 \cdots inst'_m$ `};`. Les instructions $inst_1$ à $inst_n$ sont exécutées si la valeur de l'expression *expr* est *non nulle*. Si cette valeur est nulle et qu'une branche `else` est spécifiée, alors les instructions $inst'_1$ à $inst'_m$ sont exécutées.

2.1.2 Les expressions.

Une expression est composée de variables et de constantes numériques reliées par des opérateurs et fonctions mathématiques. Il est aussi possible d'utiliser les parenthèses `(` et `)`.

Opérateurs arithmétiques. Il s'agit des opérateurs `+`, `-`, `*`, `/`, puissance `**` (fonction C `pow`) et modulo `%` (fonction C `fmod`). Les règles de priorité et d'associativité classiques sont utilisées. De plus, l'opérateur `-` peut être utilisé avec un seul argument pour calculer l'opposé d'un nombre.

Fonctions mathématiques. Sont également autorisées dans les expressions quelques fonctions C de la bibliothèque mathématique : le sinus `sin`(*exp*) et le cosinus `cos`(*exp*) (en radian), la fonction à deux arguments `atan2`(exp_y, exp_x) (qui renvoie l'angle, en radians, que fait un point avec l'origine) et la fonction `floor`(*exp*) (arrondi à l'entier immédiatement inférieur).

Opérateurs de comparaison. On peut également utiliser les opérateurs de comparaison `==` ($=$), `!=` ($\neq$), `>`, `<`, `>=` ($\geq$) et `<=` ($\leq$) dans les expressions. Le résultat d'une comparaison est 1 si la relation est vraie, 0 sinon. Ces opérateurs ont une priorité plus faible que celle des opérateurs arithmétiques : `X+1>2*Y` se lit `(X+1)>(2*Y)`. Attention à ne pas confondre l'opérateur de comparaison `==` avec l'instruction d'affectation `=`.

Opérateurs booléens. Enfin, les opérateurs `&&`, `||` et `!` correspondent aux fonctions booléennes suivantes : `&&` renvoie 1 si ses deux argument sont différent de 0, et 0 sinon ; `||` renvoie 1 si un au moins de ses arguments est différent de 0, et 0 sinon ; `!` renvoie 1 si son argument s'évalue en 0, 0 sinon. L'opérateur `||` a la plus basse priorité, suivi de `&&`, puis de `!`, puis des opérateurs de comparaison vus dans le paragraphe précédent. Par exemple `X>0 && Y>0 || X<0 && Y<0` se lit `((X>0) && (Y>0)) || ((X<0) && (Y<0))`.

2.2 Exécution de l'interpréteur

Chaque programme décrit une image, c'est à dire une fonction qui à chaque position $(x, y) \in \mathbb{R}^2$ du plan associe un niveau de gris de la manière suivante : le niveau de gris du point (x, y) est la valeur de la variable `C` après avoir exécuté le programme dans lequel les variables `X` et `Y` ont été initialisées à x et y.
Le niveau de gris est un flottant entre 0 et 1, 0 représentant le noir et 1 le blanc. Si `C` est supérieur à 1 on utilisera la valeur 1. Si `C` est négatif, on utilisera la valeur 0.
Pour obtenir une image de largeur l pixels et de hauteur h pixels, il est nécessaire d'exécuter le programme pour $l \times h$ valeurs différentes de (`X`,`Y`). Comme convention, les valeurs de `X` et `Y` passées au programme sont toujours entre 0 et 1 : le pixel à la position $(0, 0)$ correspondra toujours à `X=Y=0`, tandis que le pixel à la position $(l-1, h-1)$ correspondra à `X=Y=1`. Augmenter l et h correspond donc à améliorer la résolution.

Le programme doit être exécuté pour de nombreuses valeurs de `X` et de `Y` ; il n'est donc pas possible de l'exécuter "à la volée" dans les actions associées aux règles `bison` comme c'était le cas en TP. Au lieu de cela, les actions du fichier `bison` doivent construire une *représentation mémoire* du programme (on utilisera, par exemple, des listes chaînées pour représenter les blocs d'instructions et des arbres pour les expressions). On écrira également une fonction d'*évaluation* capable d'interpréter le programme en parcourant cette représentation mémoire, autant de fois que nécessaire.

Au final, le pseudo-code de l'interpréteur sera :

- construire une représentation mémoire du programme à l'aide de `flex` et `bison`
- pour chaque x dans $[0; l-1]$
  - pour chaque y dans $[0; h-1]$
    - mettre $x/(l-1)$ dans `X` et $y/(h-1)$ dans `Y`
    - évaluer la représentation mémoire du programme
    - stocker l'intensité $\max(0, \min(1, \mathtt{C}))$ dans le pixel (x, y)

2.3 Format d'images TGA

On utilisera le format d'image TGA comme format de sortie car il est reconnu par la plus part des logiciels de dessin et est assez facile à générer à partir d'un programme C. Une image de `l` pixels en largeur sur `h` pixels en hauteur est encodée par une fichier contenant :

- un *header* de taille 18 octets de long contenant (en comptant les positions à partir de 0) :
  - le nombre 2 à la position 2,
  - l'octet de poids faible de la largeur `l % 256` à la position 12,
  - l'octet de poids fort de la largeur `l / 256` à la position 13,
  - l'octet de poids faible de la hauteur `h % 256` à la position 14,
  - l'octet de poids fort de la hauteur `h / 256` à la position 15,
  - le nombre 24 (0x18 en hexadécimal) à la position 16,
  - des 0 partout ailleurs ;
- l'image proprement-dite composée de `h` lignes de `l` colonnes de triplet d'octets ; chaque triplet représente la couleur d'un pixel par son taux de bleu, puis de vert, puis de bleu ; chaque taux étant codé par un octet, il peut prendre une valeur entre 0 (la composante n'apparaît pas) et 255 (la composante apparaît à son maximum d'intensité).

Exemple. Une image de trois pixels sur trois pixels représentant le drapeau bleu-blanc-rouge sera codée par la suite d'octets en hexadécimal suivante :

```
0x00 0x00 0x02 0x00 0x00 0x00 0x00 0x00   0x00 0x00 0x00 0x00 0x03 0x00 0x03 0x00
0x18 0x00 0xff 0x00 0x00 0xff 0xff 0xff   0x00 0x00 0xff 0xff 0x00 0x00 0xff 0xff
0xff 0x00 0x00 0xff 0xff 0x00 0x00 0xff   0xff 0xff 0x00 0x00 0xff
```

Remarques. Une documentation plus complète du format TGA est disponible à l'adresse : `http://www.ludorg.net/amnesia/TGA_File_Format_Spec.html`. Le format TGA support plusieurs sous-formats d'images : compressé / non-compressé, avec ou sans palette, etc. Le format décrit ici correspond à une *image RGB non compressée* ; c'est le format le plus simple.

2.4 Exemples

Voici deux autres exemples de programmes :

```
/* Échiquier 2 */
XX = ((X-0.5)**2 + (Y-0.5)**2) ** 0.5 ;
YY = 3.14 + atan2( Y-0.5, X-0.5) ;
C = (floor(XX*10) + floor(YY*10)) % 2 ;
```

```
/* Spirale 2 */
XX = (X-0.5) * 30 ;
YY = (Y-0.5) * 30 ;
RR = (XX**2 + YY**2) ** 0.5 ;
X2 = XX * cos(RR) - YY * sin(RR) ;
C = X2 / 10 ;
```

À vous d'en imaginer de nouveaux !

3 Extensions

Cette partie propose quelques extensions que vous pouvez ajouter si vous avez le temps. Vous pouvez traiter une ou plusieurs extension(s) (pas forcément dans l'ordre proposé ici) mais elles ne sont pas obligatoires. Vous pouvez également proposer une ou des extension(s) de votre cru !
Il vaut mieux se limiter à un interpréteur sans extension mais qui marche bien que d'essayer de tout faire... mal !

3.1 Images en couleurs

Plutôt que de calculer l'intensité `I` de chaque pixel, on peut décider de calculer un taux de rouge, de vert, et de bleu entre 0 et 1 dans des variables `R`, `V`, `B`. On obtient ainsi des images en couleurs !

Exemple.

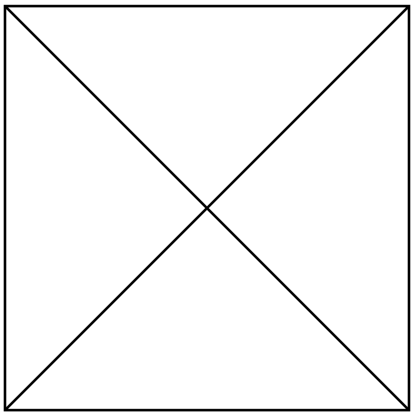
```
R = 0.5 + 0.5*sin(X*12+Y*16+1) + 0.3*sin(23*X+Y*19) ;
G = 0.5 + 0.5*sin(X*13-Y*13+2) + 0.3*sin(29*X+Y*35+1) ;
B = 0.5 + 0.5*sin(13*X-11*Y-1) + 0.3*sin(31*X+Y*18+2) ;
```

3.2 Boucles

Une boucle s'écrit `while (`*expr*`) {` $inst_1 \cdots inst_n$ `} ;` et exécute les instructions $inst_1$ à $inst_n$ tant que la condition *expr* est vraie. Juste après l'exécution de $inst_n$, l'expression *expr* est à nouveau évaluée et, si le résultat est non nul, l'exécution reprend à $inst_1$. Si le résultat est 0, alors l'exécution continue à l'instruction suivant le bloc d'instructions. Si la condition est initialement fausse, le bloc n'est pas exécuté.

Exemple. Ensemble de Mandelbrot.

```
I=0 ; ZX=0 ; ZY=0 ;
XX = 2*X-1.5 ; YY = 2*Y-1 ;
while (I < 40 && ZX**2 + ZY**2 < 40) {
  T = ZX**2 - ZY**2 + XX ;
  ZY = 2*ZX*ZY + YY ;
  ZX = T ;
  I = I +1 ;
} ;
C = 1 - I/40 ;
```



3.3 Fonctions utilisateurs

L'instruction `fun` *nom*`(`var_1`,...,`var_n`) {` $inst_1 \cdots inst_m$ `} ;` définit une nouvelle fonction de nom *nom*, ayant comme argument les noms de variables var_1 à var_n et pour corps le bloc d'instructions $inst_1$ à $inst_m$. Une telle fonction peut retourner une valeur en affectant une valeur à la variable spéciale `RETOUR`.
Un appel de fonction *nom*`(`$expr_1$`,...,` $expr_n$`)` peut alors être utilisé dans une expression : les expressions $expr_1$ à $expr_n$ sont alors évaluées et leur valeur stockées dans les variables arguments var_1 à var_n, puis le corps de la fonction est exécuté. Quand la fonction retourne, les variables argument var_1 à var_n reprennent la valeur qu'elles avaient avant l'appel (ces variables sont *locales*). L'évaluation de l'expression continue en utilisant la valeur retournée par la fonction via la variable `RETOUR`.

Exemple.

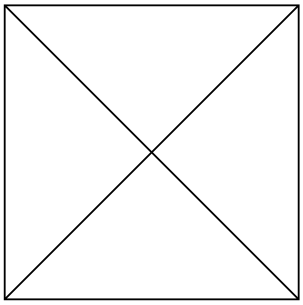
```
fun ring(XX,YY) {
  RETOUR = floor(((X2+XX)**2 + (Y2+YY)**2) ** 0.5) ;
} ;
X2 = (X-0.5)*10 ; Y2 = (Y-0.5)*10 ;
C = (ring(-1,1) + ring(1,1) + ring(0,-0.33)) % 2 ;
```

3.4 Transformations d'images

Dans cette extension, on donne à l'interpréteur, en plus d'un programme, une image en entrée. Pour permettre au programme d'accéder aux informations contenues dans l'image, on ajoute une fonction `img` qui prend deux expressions en argument : une abscisse x et une ordonnée y entre 0 et 1 et renvoie l'intensité du pixel à la position $(\lfloor x * (l'-1)\rfloor, \lfloor y * (h'-1)\rfloor)$ où l' et h' sont la largeur et la hauteur, en pixel, de l'image d'entrée. Par exemple, `img`$(0.5, 0.5)$ est l'intensité du pixel milieu de l'image. Si un des argument n'est pas entre 0 et 1, on renvoie la couleur noire 0.

Exemple. Vagues horizontales.

```
XX = 1 + X + 0.05*sin(30*Y) ;
C = img( XX % 1, Y ) ;
```



$\Longrightarrow$