Projet - Interpréteur de Commandes de Dessin

(Version du 19 mars 2018)

À rendre le vendredi 4 mai, 12h

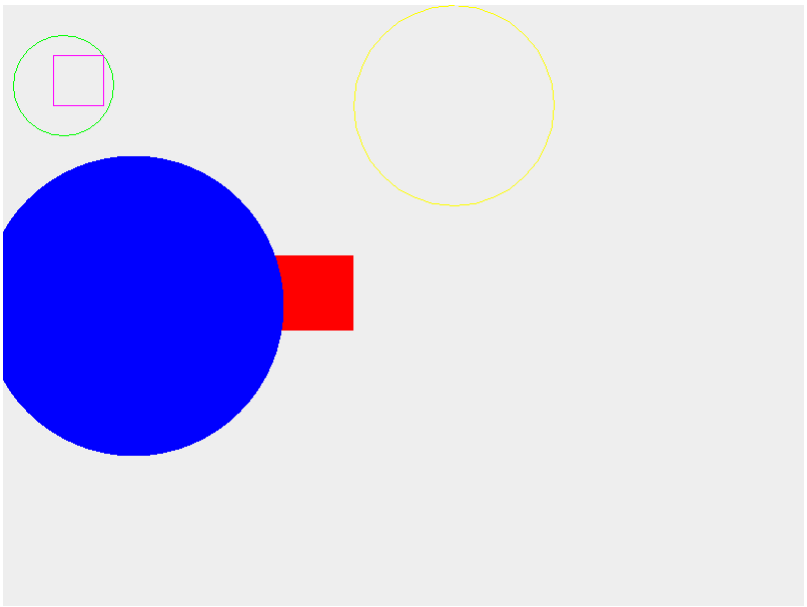
1 Introduction

Il existe aujourd'hui une multitude de logiciels permettant de faire du dessin numériquement (comme Inkscape, GIMP, ou Krita, pour ne citer que les logiciels libres). Cependant, tracer précisément (au pixel près) des formes géométriques se révèle être une tâche particulièrement complexe sur la plupart des logiciels, qui demande souvent des dizaines d'opérations pour obtenir le résultat voulu.

Dans ce projet, vous implémenterez un interpréteur permettant de tracer automatiquement et précisément des formes géométriques simples, en lisant des instructions dans un fichier. Voici un exemple simple d'un tel fichier :

```
FillRect(250,250,100,75,#FF0000);
Begin
  DrawCircle(60,80,50,#00FF00);
  FillCircle(130,300,150,#0000FF);
  DrawCircle(450,100,100,#FFFF00);
End;
DrawRect(50,50,50,50,#FF00FF);
```

Ce fragment de code doit produire le dessin suivant :



1.1 Instructions de codage

Vous devrez utiliser JFLEX pour l'analyse lexicale. Votre analyseur syntaxique doit construire un arbre de syntaxe abstraite, que votre programme va interpréter dans la suite pour créer le dessin.

Vous trouverez avec ce sujet une base de code dans le fichier **Main.java**. Ce fichier contient une classe `Main` qui initialise la fenêtre et y attache une instance de la classe `MyCanvas` qui permet d'y dessiner. Vous trouverez aussi quelques fichiers de test (bons ou mauvais) qui vous permettront de vérifier que votre programme fonctionne correctement.

Le dessin devra être entièrement réalisé en utilisant le paquet `java.awt.Graphics2D`, en particulier les fonctions `drawOval`, `fillOval`, `drawRect`, `fillRect` et `setColor`. **Il est fortement recommandé de lire la documentation associée.**

Vous devrez appeler votre interpréteur depuis la méthode `paintComponent` de la classe `MyCanvas` et lui passer l'objet de type `Graphics2d` afin qu'il puisse y dessiner. Ici, le constructeur de la classe `MyCanvas` récupère le nom du fichier passé en premier argument au programme, ce qui vous permettra d'y accéder dans cette classe et de passer le fichier à votre analyseur. Une possibilité d'implémentation est donnée en commentaire dans la classe `MyCanvas`.

La commande suivante

```
java Main dessin
```

doit exécuter votre programme sur le fichier `dessin` et produire le dessin correspondant dans la fenêtre.

Des fichiers exemple seront mis à votre disposition sur Moodle.

1.2 Organisation

Votre projet devra être rendu sur Moodle sous la forme d'une archive comprenant tous les fichiers nécessaires au projet ainsi qu'un fichier README précisant les parties réalisées, les extensions implémentées, ainsi que la marche à suivre pour le compiler.

Vous êtes **fortement encouragés à réaliser ce projet en binôme**. Les monômes sont tolérés (mais ils seront notés de la même façon que les binômes) ; les trinômes (et plus) sont **strictement interdits**. Vous pouvez former un binôme avec un.e étudiant.e d'un autre groupe de TD si vous le souhaitez.

Il y aura une soutenance à une date qu'on va vous annoncer plus tard. La note de projet sera basée sur le code rendu, et la qualité de la soutenance. La notation des deux membres d'un binôme est individuelle. **Tout plagiat sera sévèrement sanctionné**.

2 Langage de Base

On s'intéresse tout d'abord à un langage simple ne contenant que des instructions de base, des expressions arithmétiques, ainsi que des blocs de code. Afin de faciliter l'analyse du language et d'éviter les ambiguïtés, toutes les expressions arithmétiques seront intégralement parenthésées. De plus, les instructions seront toutes terminées par des point-virgules.

Votre mission est d'écrire un analyseur lexical et syntaxique permettant de traiter ce langage, de produire un arbre de syntaxe abstraite (AST), puis d'écrire un interpréteur produisant le dessin voulu à partir de l'AST.

Jetons. Afin d'analyser le langage, vous aurez d'abord besoin de le séparer en jetons (tokens). Ceux-ci peuvent être séparés en plusieurs catégories : les nombres, les symboles (opérateurs et parenthèses), les mots-clés, et les codes couleurs. Tous les mots-clés commenceront par une lettre majuscule (afin de les distinguer plus facilement des variables dans la suite), et les codes couleurs commenceront par un `#` afin de les distinguer des nombres normaux.

En particulier, la syntaxe des nombres, des couleurs, et des opérateurs est donnée par les expressions suivantes :

```
   nombre → [0-9]+
      hex → [0-9A-F]
  couleur → #{hex}{hex}{hex}{hex}{hex}{hex}
opérateur → "+" | "-" | "/" | "*"
```

Un exemple de code couleur est donc `#A015FF`, où `A0` est la valeur en hexadécimal de la composante rouge (160), `15` celle de la composante verte (21), et `FF` celle de la composante bleue (255).

Note : Afin de convertir un code hexadécimal en entier, vous pouvez utiliser la fonction `Integer.parseInt` en passant `16` comme deuxième argument. Par exemple, l'instruction `Integer.parseInt("A2",16)` s'évalue en `18`.

Grammaire. La grammaire du langage est la suivante, les non-terminaux sont en gras et en italiques, les terminaux (jetons) en police `code` :

programme	→	***suite-instructions***
instruction	→	`Begin` ***suite-instructions*** `End`
	|	`DrawCircle (` ***expr*** `,` ***expr*** `,` ***expr*** `, couleur )`
	|	`FillCircle (` ***expr*** `,` ***expr*** `,` ***expr*** `, couleur )`
	|	`DrawRect (` ***expr*** `,` ***expr*** `,` ***expr*** `,` ***expr*** `, couleur )`
	|	`FillRect (` ***expr*** `,` ***expr*** `,` ***expr*** `,` ***expr*** `, couleur )`
suite-instructions	→	***instruction*** `;` ***suite-instructions*** | ε
expr	→	`nombre` | `(` ***expr*** `opérateur` ***expr*** `)`

Interprétation. Votre langage doit interpréter les instructions de base `DrawCircle`, `FillCircle`, `DrawRect` et `FillRect`. Leur interprétation est la suivante :

- `DrawCircle(x,y,z,col)` affiche un cercle de centre `(x,y)`, de rayon `z` et de couleur `col`.
- `FillCircle(x,y,z,col)` affiche un disque de centre `(x,y)`, de rayon `z` et de couleur `col`.
- `DrawRect(x,y,w,h,col)` affiche un rectangle dont le coin supérieur-gauche et situé en `(x,y)`, de longueur `w`, de hauteur `h` et de couleur `col`.
- `FillRect(x,y,w,h,col)` affiche un rectangle *plein* dont le coin supérieur-gauche et situé en `(x,y)`, de longueur `w`, de hauteur `h` et de couleur `col`.

L'origine du repère associé à la fenêtre (position `(0,0)`) est considérée en haut à gauche, l'axe des abscisses étant orienté vers la droite et l'axe des ordonnées vers le bas.

Erreurs. Votre programme devra traiter correctement les erreurs pouvant survenir pendant l'exécution, et fournir des messages explicites avant de terminer. En particulier, les messages doivent contenir la position de l'erreur, ainsi que les informations pertinentes (comme le jeton problématique en cas d'erreur d'analyse syntaxique).

3 Constantes

On souhaite maintenant ajouter des constantes au langage. Pour cela, on ajoute les règles suivantes à la grammaire :

instruction	→	`Const identificateur =` ***expr***
expr	→	`identificateur`

Où le jeton `identificateur` représente les mots commençant par une minuscule et pouvant contenir des "_". Formellement, il est défini de la façon suivante :

`identificateur` → `[a-z][a-zA-Z_]*`

La suite d'expressions suivante doit créer une constante `pos_x` de valeur `10`, puis créer une constante `pos_y` de valeur `25` :

```
Const pos_x = 10;
Const pos_y = ((pos_x * 2) + 5);
```

Les constantes ne contiendront que des valeurs entières. De plus, comme leur nom l'indique, leur valeur ne pourra pas changer au cours de l'exécution de l'AST. Ainsi, vous pouvez par exemple utiliser un objet de type `HashMap<String,Integer>` pour stocker les valeurs associées à chaque constante.

Remarque. La portée des constantes devra être limitée *à l'intérieur du bloc* dans lequel elles sont définies. Ainsi, le programme suivant doit renvoyer une erreur signifiant que la constante `pos_x` n'est pas définie à la ligne 5 (et pas avant) :

```
Begin
    Var pos_x = 10;
    DrawCircle(pos_x, 50, 10, #00FF00);
End
DrawCircle(pos_x, 100, 5, #FF0000);
```

Vous ferez également attention à ce que les constantes déclarées dans un bloc n'écrasent pas les constantes ayant le même nom déclarées dans un bloc plus externe.

4 Conditionnelles

Dans cette partie, on souhaite ajouter des expressions conditionnelles au langage. Pour cela, on ajoute cette fois la règle suivante à la grammaire :

instruction → `If` ***expr*** `Then` ***instruction*** `Else` ***instruction***

Une expression de la forme `If e Then ins1 Else ins2` doit évaluer l'instruction `ins1` si l'expression `e` s'évalue en une valeur différente de `0`, et évaluer `ins2` sinon.

5 Extension : Variables

Si vous avez fini les parties précédentes, vous pouvez maintenant essayer d'ajouter des variables au langage. Pour cela, il faut ajouter les règles suivantes à la grammaire :

instruction → `Var identificateur =` ***expr***
| `identificateur =` ***expr***

La suite d'expressions suivante doit créer une variable `pos_x`, l'initialiser à la valeur `10`, puis mettre sa valeur à `25` :

```
Var pos_x = 10;
pos_x = ((pos_x * 2) + 5);
```

Contrairement aux constantes, la valeur des variables peut changer au cours de l'exécution, comme indiqué par l'instruction d'affectation. Une simple structure de type `HashMap<String,Integer>` ne suffit donc plus.

Cette partie est délibérément laissée ouverte et vous êtes libres d'utiliser la structure que vous désirez. Cependant, comme pour les constantes, vous ferez particulièrement attention aux problèmes de portée et d'écrasement des variables précédemment définies.

6 Extensions Libres

Si vous avez terminé les trois parties précédentes, vous pouvez ajouter diverses extensions à votre langage. En voici quelques suggestions, mais vous pouvez évidemment en faire d'autres si vous le désirez :

- Le langage ne permet actuellement pas de répéter aisément un motif sans copier-coller des dizaines de lignes de code, ce qui peut se montrer très laborieux. Pour résoudre ce problème, vous pouvez ajouter des boucles de la forme `While` ***expr*** `Do` ***instruction*** par exemple.
- Il est pour le moment impossible d'écrire des expressions conditionnelles ne contenant pas de branche `"Else"`. Etendez la grammaire afin de permettre des expressions de ce type. Une telle grammaire étant ambiguë, vous devrez en particulier réfléchir à une stratégie pour résoudre les ambiguïtés [1].

1. Voir le "dangling else problem" (https ://en.wikipedia.org/wiki/Dangling_else)