Aabu

Tesselz : un langage pour générer des figures répétitives en SVG

8 décembre 2020

Table des matières

J'ai eu envie de concevoir un langage dédié à la description de figures géométriques répétitives. C'est un projet à titre d'exercice, pour apprendre à concevoir un langage et un interpréteur (et avoir une excuse pour approfondir mon expérience avec Rust).

Ce billet présente le résultat de ce petit projet-jouet: **Tesselz** , un langage impératif interprété qui permet de construire des figures géométriques répétitives faites de polygones et générer un rendu SVG.

1. Démonstration

Voici deux exemples pour montrer à quoi ressemble le langage et ce qu'il génère.

1.1. Papier peint

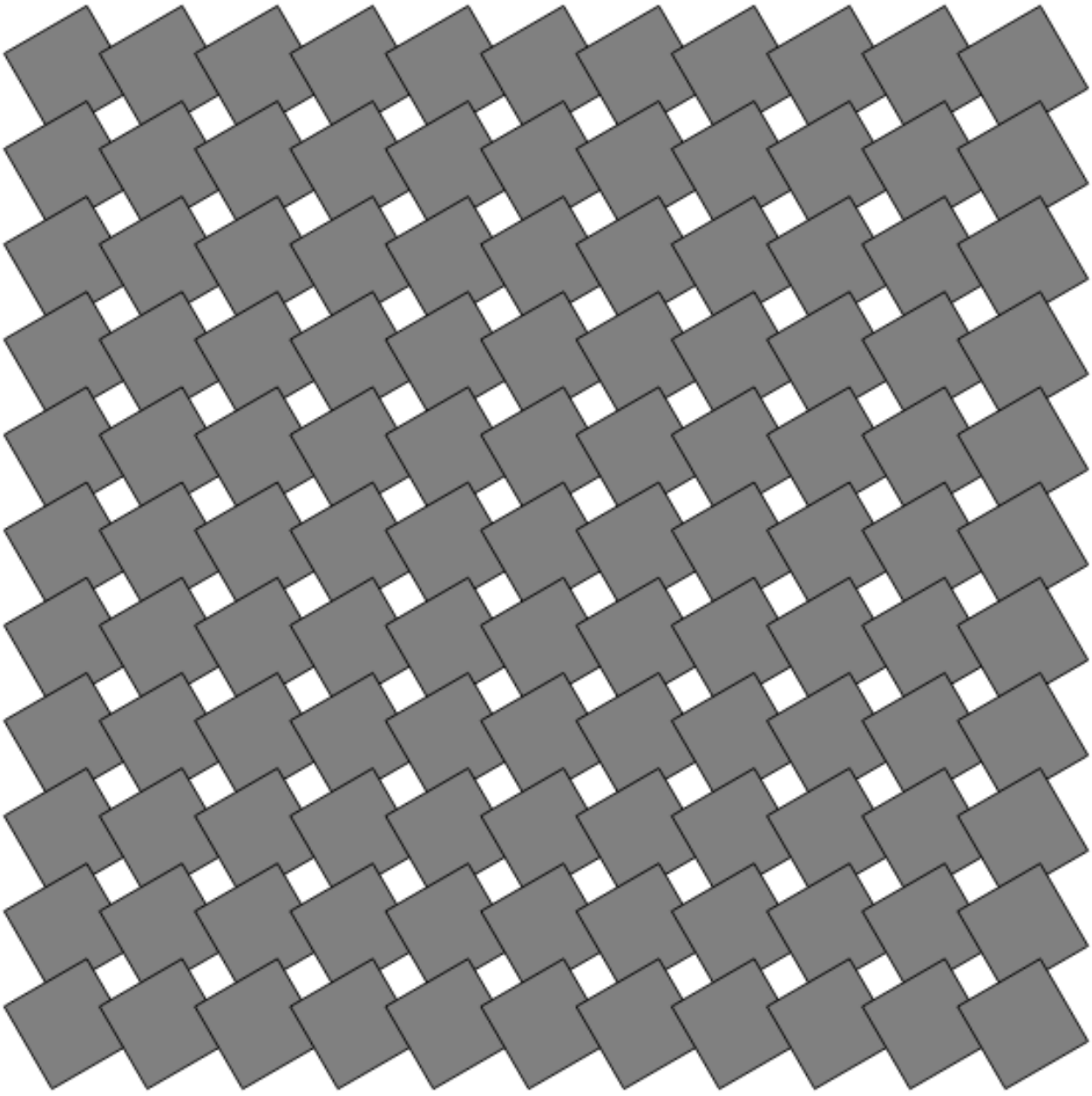

FIGURE 1.1. – Papier peint

```
k = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
i = mul(k, vector(50, 0));
j = mul(k, vector(0, 50));
vectors = add(i, j);
A = point(0, 0);
B = point(50, 0);
C = point(50, 50);
```

```
D = point(0, 50);
square = polygon({A, B, C, D});
angle = div(3.14, 3);
square_rot = rotate(square, angle, point(25, 25));
pattern = translate(square_rot, vectors);
pattern > "output.svg";
```

1.2. Squared circle

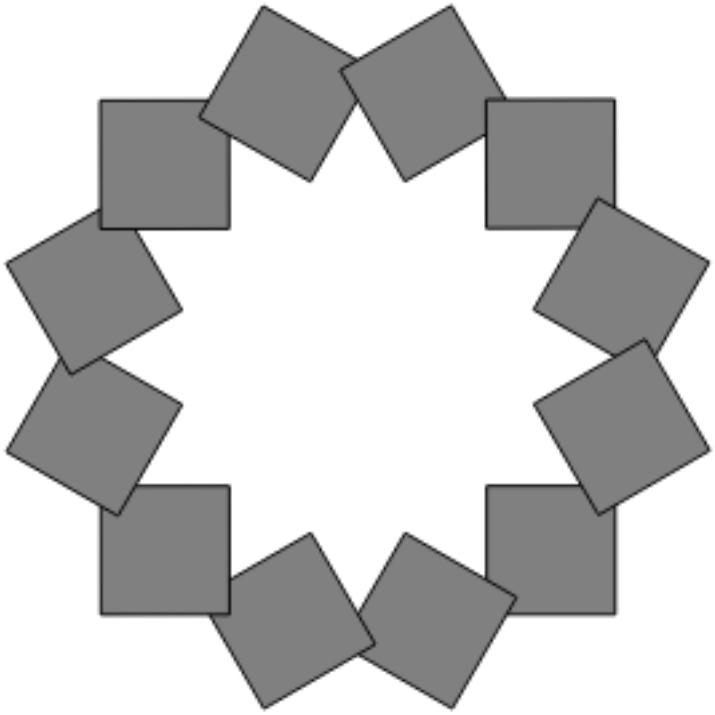

FIGURE 1.2. – *Squared circle*

```
k = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
A = point(0, 0);
B = point(50, 0);
C = point(50, 50);
D = point(0, 50);
square = polygon({A, B, C, D});
square_offset = translate(square, vector(550, 550));
angles = mul(div(6.28, 12), k);
pattern = rotate(square_offset, angles, point(500, 500));
pattern > "output.svg";
```

2. Description du langage

Le langage est volontairement simpliste. Il contient juste ce qu'il faut pour décrire des polygones et des transformations géométriques basiques et sortir un rendu SVG. La conception se fonde sur un plan muni d'un repère cartésien pour décrire les formes et transformations.

2.1. Instructions

Un programme est une séquence d'*instructions* terminées par des points-virgules. Il y a seulement deux types d'instructions:

- **Assignment**: `var = expr;` évalue l'expression `expr` et associe l'identifiant `var` au résultat.
- **Render**: `expr > "out.svg";` évalue l'expression `expr` et écrit un fichier `out.svg` contenant le rendu SVG de l'expression.

2.2. Objets

2.2.1. Éléments

Le langage connaît les types d'*objets élémentaires* suivants:

- **Number**: un nombre flottant;
- **Vector**: un couple de nombres flottants;
- **Point**: encore un couple de nombres flottants (mais sémantique différente);
- **Polygon**: un tuple de points.

2.2.2. Ensembles

Les objets peuvent être groupés dans des *objets ensemble* de type **Set**. Les ensembles sont en fait ordonnés, et portent donc mal leur nom. Les ensembles peuvent contenir n'importe quel objet, y compris d'autres ensembles.

2.3. Expressions

Les objets sont obtenus en évaluant des *expressions*. Les types d'expressions sont les suivants:

- **Number**: un nombre flottant littéral, comme dans la plupart des langages, par exemple `3.14`;
- **Identifier**: une étiquette désignant un objet (créée par une affectation), par exemple `pointA`;
- **FunctionCall**: un nom de fonction suivi par une liste d'expressions entre parenthèses séparés par des virgules, par exemple `mul(a, b)`;
- **Set**: une liste d'expressions entre accolades séparées par des virgules, par exemple `{a, b, c}`.

2.4. Fonctions

2.4.1. Listes des fonctions

Le langage dispose des fonctions suivantes.

- Opérations élémentaires
  - `add(_, _)`: ajoute des nombres ou des vecteurs;
  - `sub(_, _)`: soustrait des nombres ou des vecteurs;
  - `mul(_, _)`: multiplie des nombres ou des nombres par des vecteurs;
  - `div(_, _)`: divise des nombres.
- Création d'objets élémentaires
  - `point(_, _)`: crée des points à partir de nombres;
  - `vector(_, _)`: crée des vecteurs à partir de nombres
  - `polygon({_, _, ...})`: crée un polygone à partir d'un ensemble (ordonné) de points.
- Transformations géométriques:
  - `translate(_, _)`: translate des formes géométriques (points ou polygones) par des vecteurs.
  - `rotate(_, _, _)`: tourne des formes géométriques (points ou polygones) d'angles donnés autour d'un point.

2.4.2. *Broadcast*

Le *broadcast* (faute d'un meilleur nom) est probablement la fonctionnalité la plus atypique du langage, même si on la retrouve tout de même régulièrement ailleurs. L'idée du *broadcast* est que les fonctions sont capables de gérer des ensembles (ordonnés…) en entrée et de sortir un ensemble mappé.

Par exemple:

- Pour des nombres `x` et `y`, `point(x, y)` est le point de coordonnées (x, y).
- Pour un ensemble `s` et un nombre `y`, `point(s, y)` est l'ensemble {point(x, y) | x s}.
- Pour un nombre `x` et un ensemble `s`, `point(x, s)` est l'ensemble {point(x, y) | y s}.
- Pour des ensembles `s1` et `s2`, `point(s1, s2)` est l'ensemble {point(x, y) | (x, y) s1 x s2}.

Avec ça, on peut exprimer des opérations répétitives sans boucle.

3. Implémentation

Le code est disponible sur GitHub . Il est écrit en Rust, avec l'aide du la bibliothèque `lalrpop` pour l'analyse lexicale et syntaxique.

Le flot global d'exécution est le suivant:

- lecture du code source d'entrée;
- analyse lexicale et syntaxique;
- interprétation du programme.

La première étape n'a rien de fondamentalement intéressant, si ce n'est qu'elle n'est pas du tout programmée de manière ergonomique.

3.1. Analyse lexicale et syntaxique

L'analyse lexicale est faite avec le *lexer* par défaut de `lalrpop`. Je n'ai pas creusé plus que ça, mais il génère les *tokens* en ignorant les espaces et autres caractères invisibles, ce qui me convenait bien.

L'analyseur syntaxique est généré par `lalrpop` à l'aide d'un fichier de grammaire (sobrement appelé `parser.lalrpop` dans mon cas), qui décrit les règles du langage. La syntaxe ressemble à du Rust, mais n'en est pas vraiment. J'en mets un extrait ci-dessous. On remarque l'imbrication des différentes règles.

```
StatementBody : Statement = {
    <a:Assignment> => Statement::Assignment(a),
    <r:Render> => Statement::Render(r),
};

Assignment: Assignment = {
    <i:Identifier> "=" <e: Expression> => Assignment {ident: i,
        expr: e},
};

Render: Render = {
    <e:Expression> ">" <f: Filename> => Render { expr: e,
        filename:f },
};

Expression: Expression = {
    <i:Identifier> => Expression::Ident(i),
    <n:Number> => Expression::Number(n),
    <c:FunctionCall> => Expression::FunctionCall(c),
    <s:Set> => Expression::Set(s),
};
```

3.2. Interprétation

3.2.1. Exécution d'un programme

L'interpréteur fonctionne en exécutant les lignes les unes à la suite des autres, tout en mettant à jour un *contexte* (concrètement une *hashmap*) pour garder trace des identifiants connus.

Au début de l'exécution, le contexte est chargé avec toutes les fonctions *builtins* du langage. L'interpréteur y aura accès par leur nom pendant l'évaluation des expressions.

Lorsque l'interpréteur exécute une instruction d'affectation, il évalue l'expression puis met à jour le contexte en ajoutant une paire (clé, valeur) liant l'identifiant et le résultat de l'évaluation (un objet).

Lors de l'exécution d'une instruction de rendu, l'interpréteur évalue l'expression et écrit le fichier correspondant en analysant l'objet retourné en termes de polygones SVG.

3.2.2. Évaluation d'expression

L'exécution du programme n'est que la structure de haut niveau de l'interpréteur. L'essentiel des calculs se passe lors de l'évaluation des expressions.

Les expressions sont évaluées par simplification récursive:

- pour un ensemble, on évalue les expressions membres et on descend ainsi récursivement,
- pour un appel de fonction, on évalue les arguments avant d'appeler la fonction,
- pour une expression simple (identifiant ou littéral), on va chercher ou on construit l'objet correspondant.

3.2.3. Fonctions builtins

Les fonctions *builtins* sont toutes programmées de manière similaire (et c'est très répétitif). L'idée générale est qu'une fonction prend en compte une liste d'objets, fait quelque chose avec et renvoie un objet.

La grande majorité des opérations significative sont faites par les fonctions, qui savent gérer différent types d'objets, opérer sur des ensembles, etc.

4. Post mortem

Après la clôture de ce petit projet, j'ai quelques remarques sur ce que j'aurai pu ou voulu faire différemment sur ce projet.

4.1. Conception du langage

4.1.1. Produit cartésien

Le *broadcast* est un peu maladroit, parce que c'est à chaque fonction de le gérer correctement. Par exemple l'implémentation de la fonction *rotate* ne permet pas le *broadcast* sur le pivot de la rotation.

Avec un produit cartésien à proprement parler, il y aurait possibilité de gérer toutes les fonctions de la même manière. Le programmeur se chargerait de constituer un ensemble de tuples et la machinerie s'occuperait de mapper la fonction appelée sur cet ensemble.

4.1.2. Union d'ensembles

Ça manque un peu pour créer des arrangements plus complexes. C'est assez facile à faire, mais chaque projet-jouet doit avoir une fin. 🍊

4.1.3. Opérateurs infixes

J'ai fait l'impasse dessus parce que c'est un peu plus compliqué à mettre en œuvre, mais ça serait vraiment plus pratique et lisible que des appels de fonctions, en particulier pour toutes les opérations usuelles du type `add` ou `mul`.

4.1.4. Autres remarques en vrac

J'aurais préféré faire des instructions terminées par un retour à la ligne *à la Python*, pour éviter du bruit visuel.

Aussi, j'ai hésité entre des appels de fonction à la OCaml (arguments séparés par des espaces, pas de parenthésage global) et la solution actuelle. La solution actuelle me paraissait un peu plus facile à implémenter et c'est donc ce que j'ai choisi.

Encore inspiré par OCaml, j'aurais aimé pouvoir traiter facilement des fonctions d'ordre supérieur et faire plein de compositions de transformations. Un peu trop complexe en regard du plaisir, donc j'ai passé mon tour.

Aussi le nom *ensemble* est mal choisi, vu qu'il s'agit plutôt de listes.

Le langage mériterait aussi une documentation plus complète, mais ça ne m'intéresse pas plus que ça de l'écrire désormais.

4.2. Implémentation

Il y a *beaucoup* de copie de données dans mon code, par facilité. Si je devais reprendre ce code, c'est un point sur lequel je travaillerai, parce qu'il doit être assez hautement inefficace en l'état.

D'ailleurs l'implémentation est vraiment spartiate, puisqu'elle ne gère en fait pas la lecture de fichiers en entrée. Le code source est écrit *en dur* dans celui de l'interpréteur. C'est facile à corriger, mais ce ne sera pas fait.

L'implémentation ne gère pas l'entrée correctement, mais la sortie non plus. Le SVG est écrit peu subtilement à coup de `print` directement dans la gestion de l'instruction de rendu dans l'interpréteur.