

Projet Draw++

Guillarme Arno

Otero Clement

Zalegh Riyad

12 janvier 2025

Table des matières

| | | |
|----------|------------------------------------------------------------------------------|-----------|
| 1 | La syntaxe du langage | 5 |
| 1.1 | Déclarations de variables | 5 |
| 1.2 | Déclarations de curseurs | 6 |
| 1.3 | Structures de contrôle | 7 |
| 1.3.1 | Conditions if-else | 7 |
| 1.3.2 | Boucle while | 8 |
| 1.3.3 | Boucle for | 9 |
| 1.4 | Manipulation des curseurs | 10 |
| 1.4.1 | Modification de la couleur | 10 |
| 1.4.2 | Rotation du curseur | 10 |
| 1.5 | Dessin de formes | 10 |
| 1.5.1 | Ligne | 10 |
| 1.5.2 | Cercle | 10 |
| 1.5.3 | Carré | 10 |
| 1.5.4 | Point | 10 |
| 1.5.5 | Arc | 10 |
| 2 | Le Lexer : Analyseur Lexical pour Draw++ | 11 |
| 2.1 | Introduction au Lexer | 11 |
| 2.2 | Fonctionnalités du Lexer | 11 |
| 2.3 | Structure du Lexer | 11 |
| 2.3.1 | Définition des Patterns Lexicaux | 11 |
| 2.3.2 | Analyse et Génération des Tokens | 12 |
| 2.3.3 | Gestion des Erreurs | 12 |
| 2.4 | Exemple d'Analyse Lexicale | 12 |
| 3 | Le Parser : Analyseur Syntaxique | 13 |
| 3.1 | Qu'est-ce qu'un Parser ? | 13 |
| 3.2 | Fonctionnement du Parser | 13 |
| 3.3 | Exemple d'Analyse : Fonction <code>parse_cursor_declaration</code> | 13 |
| 3.4 | Importance de l'AST | 14 |
| 4 | L'Interpréteur | 15 |
| 4.1 | Rôle de l'Interpréteur | 15 |
| 4.2 | Fonctionnement Général | 15 |
| 4.3 | Exemple d'Utilisation : <code>check_set_position</code> | 15 |
| 4.4 | Analyse de la Fonction | 16 |
| 4.5 | Exemple d'AST et Résultat | 16 |
| 4.6 | Importance de l'Interpréteur | 16 |
| 5 | Le Compiler : Traduction en C | 17 |
| 5.1 | Le Compilateur de Draw++ | 17 |
| 5.2 | Rôle du Compilateur dans le Projet Draw++ | 17 |
| 5.3 | Fonctionnement du Compilateur | 17 |

| | | |
|----------|----------------------------------------------------|-----------|
| 6 | L’IDE de Draw++ : Aperçu et Fonctionnalités | 18 |
| 6.1 | Introduction | 18 |
| 6.2 | Fonctionnalités principales | 18 |
| 7 | Conclusion | 26 |

Introduction

Draw++ est un langage de programmation spécialement conçu pour la création graphique, visant à rendre l'interaction avec des éléments visuels simple et intuitive. Grâce à un ensemble d'instructions claires et accessibles, il permet aux utilisateurs de dessiner facilement des formes et de créer des illustrations graphiques.

Le projet ne se limite pas au langage lui-même ; il intègre également un environnement de développement intégré (IDE) pensé pour accompagner les utilisateurs à chaque étape du processus. Cet IDE facilite l'écriture, le test, et le débogage du code Draw++, tout en offrant une visualisation instantanée des résultats graphiques à l'écran.

l'IDE propose une expérience de programmation fluide et intuitive. Que ce soit pour des débutants ou des utilisateurs expérimentés, Draw++ offre un cadre interactif et réactif qui encourage la créativité.

1 La syntaxe du langage

Nous allons voir dans cette partie les éléments de base et les fonctionnalités principales, allant des déclarations de variables aux structures de contrôle, en passant par les manipulations graphiques avancées.

La première étape du projet a été la définition d'une grammaire claire et conforme aux exigences du cahier des charges.

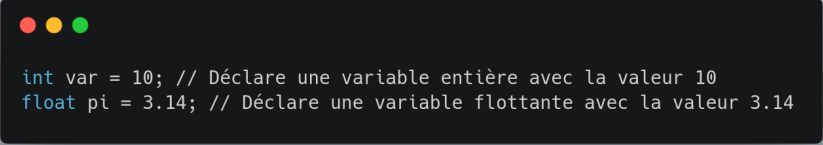
```
# grammar.py
# This file defines the grammar rules for the Draw++ language.

grammar = {
    "programme": {
        "description": "A Draw++ program is a sequence of instructions.",
        "rule": ["instruction*"] # Zero or more instructions
    },
    "instruction": {
        "description": "An instruction can represent various commands such as cursor declarations,
movements, drawings, conditions, loops, etc.",
        "rule": [
            "declaration_curseur", # Declares a cursor
            "positionnement_curseur", # Sets the cursor's position
            "couleur_curseur", # Changes the cursor's color
            "epaisseur_curseur", # Sets line thickness
            "deplacement_curseur", # Moves the cursor
            "rotation_curseur", # Rotates the cursor
            "dessin_ligne", # Draws a line
            "dessin_carre", # Draws a square
            "dessin_cercle", # Draws a circle
            "dessin_point", # Draws a point
            "dessin_arc", # Draws an arc
            "declaration_variable", # Declares a variable
            "affectation_variable", # Assigns a value to a variable
            "conditionnelle", # If-else statement
            "boucle_for", # For loop
            "boucle_while", # While loop
            "bloc_instructions", # Instruction block
            "appel_bloc", # Calls a block of instructions
            "fonction", # Defines a function
            "appel_fonction" # Calls a function
        ]
    },
    "expression": {
        "description": "Une expression qui peut être une valeur, une opération mathématique, ou une
combinaison.",
        "rule": [
            "valeur", # Une valeur seule
            "expression", "opérateur_arithmétique", "expression" # Une opération entre deux
expressions
        ]
    },
    "opérateur_arithmétique": {
        "description": "Opérateur arithmétique utilisé dans les calculs.",
        "rule": ["+", "-", "*", "/", "%"]
    },
    "affectation_variable": {
        "description": "Affecte une valeur ou une expression à une variable existante.",
        "rule": ["identifiant", "=", "expression", ";"]
    }
}
```

FIGURE 1 – Une partie de la grammaire de Draw ++.

1.1 Déclarations de variables

Les variables dans Draw++ peuvent être déclarées à l'aide des types de base tels que `int` et `float`. Chaque déclaration doit être suivie d'un point-virgule (;).




```
int var = 10; // Déclare une variable entière avec la valeur 10
float pi = 3.14; // Déclare une variable flottante avec la valeur 3.14
```

FIGURE 2 – déclaration de variables en Draw

1.2 Déclarations de curseurs

Les curseurs sont des entités graphiques centrales dans Draw++ qui servent de points de départ pour le dessin de formes et la manipulation graphique.



```
cursor mainCursor; // Déclare un curseur nommé 'mainCursor'
```

FIGURE 3 – déclaration de Curseur en Draw

1.3 Structures de contrôle

1.3.1 Conditions if-else

La structure conditionnelle permet d'exécuter un bloc de code si une condition est vraie. Un bloc `else` optionnel peut être ajouté pour traiter le cas où la condition est fausse.

Syntaxe :

```
if (<condition>) {  
    // Bloc d'instructions exécuté si la condition est vraie  
} else {  
    // Bloc d'instructions exécuté sinon  
}
```



FIGURE 4 – Boucle IF ELSE en Draw

1.3.2 Boucle while

La boucle `while` exécute un bloc d'instructions tant qu'une condition donnée reste vraie.

Syntaxe :

```
while (<condition>) {  
    // Bloc d'instructions exécuté tant que la condition est vraie  
}
```



FIGURE 5 – Boucle WHILE en Draw

1.3.3 Boucle for

La boucle `for` est utilisée pour effectuer des itérations avec une initialisation, une condition et une mise à jour à chaque tour.

Syntaxe :

```
for (<initialisation>; <condition>; <mise à jour>) {  
    // Bloc d'instructions exécuté à chaque itération  
}
```

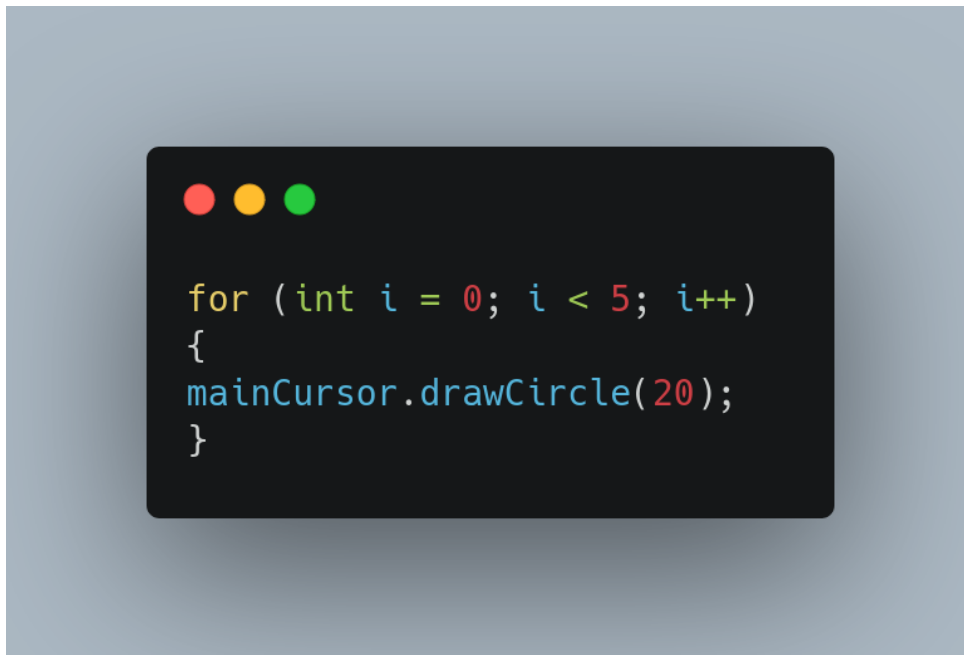


FIGURE 6 – Boucle FOR en Draw

1.4 Manipulation des curseurs

1.4.1 Modification de la couleur

```
mainCursor.setColor(red); // Change la couleur du curseur en rouge
```

1.4.2 Rotation du curseur

```
mainCursor.rotate(45); // Fait pivoter le curseur de 45 degrés
```

1.5 Dessin de formes

1.5.1 Ligne

```
mainCursor.drawLine(50); // Dessine une ligne de 50 unités
```

1.5.2 Cercle

```
mainCursor.drawCircle(30); // Dessine un cercle avec un rayon de 30 unités
```

1.5.3 Carré

```
mainCursor.drawSquare(40); // Dessine un carré de 40 unités de côté
```

1.5.4 Point

```
mainCursor.drawPoint(); // Dessine un point
```

1.5.5 Arc

```
mainCursor.drawArc(20, 90); // Dessine un arc de rayon 20 et de 90 degrés
```

Cette présentation de la syntaxe de Draw++ offre une vue d'ensemble des fonctionnalités essentielles du langage.

2 Le Lexer : Analyseur Lexical pour Draw++

2.1 Introduction au Lexer

Le Lexer (ou analyseur lexical) est une composante essentielle dans le processus de compilation ou d'interprétation d'un langage. Sa fonction principale est de convertir le code source brut en une liste structurée d'unités lexicales appelées *tokens*. Ces tokens représentent les éléments fondamentaux du langage, comme les mots-clés, les symboles, les identifiants ou les valeurs numériques.

Dans le cadre du langage Draw++, le Lexer a pour objectif de reconnaître les instructions spécifiques au dessin, les structures de contrôle, ainsi que les manipulations des curseurs. Il traduit le code source en une liste de tokens qui seront ensuite traités par le Parser.

2.2 Fonctionnalités du Lexer

- **Lecture et Conversion du Code Source**

Le Lexer prend en entrée le code source de Draw++ et l'analyse ligne par ligne, caractère par caractère, pour en extraire des *patterns* (modèles) définis dans des expressions régulières.

- **Génération des Tokens**

Chaque élément reconnu est transformé en un token, composé :

- d'un type (`TokenType`) qui identifie sa nature (par exemple, `TokenType.INT` pour un entier ou `TokenType.CURSOR` pour le mot-clé `cursor`).

- **Gestion des Erreurs Lexicales**

En cas d'élément non reconnu dans le code source, le Lexer déclenche une erreur, signalant un caractère ou une séquence invalide.

2.3 Structure du Lexer

Le fichier `lexer.py` contient une classe principale `Lexer` qui implémente l'analyse lexical. Voici les principales sections du fichier :

2.3.1 Définition des Patterns Lexicaux

La méthode `tokenize` s'appuie sur des expressions régulières pour identifier les différentes catégories de tokens.

```
token_specs = [  
    ("PLUS_PLUS", r'\+\+'),  
    ("MINUS", r'-'),  
    ("NUMBER", r'-?\d+(\.\d*)?'),  
    ("IDENTIFIER", r'[a-zA-Z_]\w*'),  
    ("SYMBOL", r'[;(){}\<>]'),  
    ("SKIP", r'[\t]+'),  
    ("NEWLINE", r'\n'),  
]
```

Chaque pattern est associé à un type de token. Par exemple :

- `NUMBER` reconnaît les nombres (entiers ou flottants).
- `IDENTIFIER` détecte les noms de variables ou curseurs.
- `SKIP` et `NEWLINE` ignorent les espaces ou les sauts de ligne.

2.3.2 Analyse et Génération des Tokens

La méthode `tokenize` parcourt le code source et applique les patterns pour créer une liste de tokens.

Chaque correspondance entre le code source et un pattern génère un nouveau token, ajouté à la liste `self.tokens`.

2.3.3 Gestion des Erreurs

Si un caractère ou une séquence n'est pas reconnue, une erreur est levée :

else:

```
raise SyntaxError(f"Unknown token '{self.source_code[pos]}' at position {pos}")
```

2.4 Exemple d'Analyse Lexicale

Prenons un extrait de code `Draw++` :

```
cursor mainCursor;
mainCursor.setPosition(10, 20);
```

Voici les étapes suivies par le Lexer :

- Reconnaissance du mot-clé `cursor` : création d'un token `TokenType.CURSOR`.
- Identification de `mainCursor` comme identifiant : token `TokenType.IDENTIFIER`.
- Reconnaissance du point-virgule : token `TokenType.SEMICOLON`.
- Reconnaissance de l'appel de méthode `setPosition` et des arguments.

Liste des tokens générés :

```
[
    Token(TokenType.CURSOR, "cursor"),
    Token(TokenType.IDENTIFIER, "mainCursor"),
    Token(TokenType.SEMICOLON, ";"),
    Token(TokenType.IDENTIFIER, "mainCursor"),
    Token(TokenType.DOT, "."),
    Token(TokenType.SET_POSITION, "setPosition"),
    Token(TokenType.LPAREN, "("),
    Token(TokenType.NUMBER, "10"),
    Token(TokenType.COMMA, ","),
    Token(TokenType.NUMBER, "20"),
    Token(TokenType.RPAREN, ")"),
    Token(TokenType.SEMICOLON, ";")
]
```

3 Le Parser : Analyseur Syntaxique

3.1 Qu'est-ce qu'un Parser ?

L'objectif principal du Parser est de convertir une séquence de tokens, produite par le lexer, en une structure hiérarchique connue sous le nom d'*arbre syntaxique abstrait* (AST). Cette structure reflète la syntaxe du code et permet de représenter logiquement le programme en fonction des règles de la grammaire du langage.

Dans le contexte de Draw++, le parser analyse le code source, vérifie sa conformité à la grammaire définie, et construit l'AST qui sera ensuite utilisé par l'interpréteur ou le compilateur.

3.2 Fonctionnement du Parser

Le parser traite chaque token dans une séquence linéaire pour construire une représentation structurée du code. Voici les principales étapes effectuées par le parser de Draw++ :

- **Initialisation** : Le parser est initialisé avec la liste des tokens produits par le lexer.
- **Parcours des tokens** : Le parser parcourt les tokens un par un et les analyse en fonction des règles de la grammaire.
- **Gestion des structures syntaxiques** : Chaque construction syntaxique (comme les boucles, les instructions, ou les déclarations) est convertie en un nœud de l'AST.
- **Gestion des erreurs** : Si une erreur syntaxique est détectée (par exemple, un point-virgule manquant ou une parenthèse non fermée), le parser l'enregistre pour que l'utilisateur puisse la corriger.
- **Construction de l'AST** : Une fois tous les tokens traités, l'AST final est renvoyé pour être utilisé par les étapes suivantes.

3.3 Exemple d'Analyse : Fonction `parse_cursor_declaration`

Cette fonction traite les déclarations de curseurs dans le code source Draw++.

```
def parse_cursor_declaration(self):
    self.expect(TokenType.CURSOR) # Assure que le token est bien un 'cursor'
    cursor_name = self.expect(TokenType.IDENTIFIER).value
    self.expect(TokenType.SEMICOLON) # Assure que la déclaration finit par un ';'
    return {"type": "CURSOR_DECLARATION", "name": cursor_name}
```

Explication : Cette fonction vérifie qu'une déclaration de curseur suit la syntaxe correcte (par exemple : `cursor mainCursor;`). Elle s'assure que :

- Le mot-clé `cursor` est présent.
- Un identifiant valide suit le mot-clé.
- La déclaration est terminée par un point-virgule (`;`).

En cas d'erreur, elle enregistre une erreur syntaxique.

3.4 Importance de l'AST

L'*arbre syntaxique abstrait* (AST) généré par le parser est une représentation hiérarchique du code source. Voici un exemple d'AST pour une déclaration simple de curseur :

Code source Draw++ :
`drawLine(50);`

AST généré :
{
 "type": "DRAW_LINE",
 "cursor": "mainCursor",
 "length": {
 "type": "VALUE",
 "value": 50
 }
}

Cet AST sera ensuite utilisé par l'interpréteur pour exécuter les instructions ou par le compilateur pour générer le code C correspondant.

4 L'Interpréteur

4.1 Rôle de l'Interpréteur

L'interpréteur est une composante essentielle de l'architecture du projet Draw++. Son rôle principal est d'analyser et d'exécuter l'*arbre syntaxique abstrait* (AST) généré par le parseur. Il vérifie la validité des nœuds syntaxiques et identifie les éventuelles erreurs de logique ou de cohérence dans le code Draw++ avant qu'il ne soit compilé.

En d'autres termes, l'interpréteur joue le rôle de "médiateur" entre la définition de l'AST et le processus de compilation final.

4.2 Fonctionnement Général

L'interpréteur parcourt l'AST nœud par nœud. Pour chaque nœud, il exécute une action spécifique selon le type du nœud :

- Validation des déclarations de variables ou de curseurs.
- Analyse des structures de contrôle telles que les boucles et les conditions.
- Gestion des erreurs lorsqu'une instruction n'est pas valide ou manque de contexte (par exemple, une variable non déclarée).

4.3 Exemple d'Utilisation : `check_set_position`

Cette fonction est appelée lorsque l'AST contient un nœud correspondant à une instruction de positionnement d'un curseur (par exemple, `mainCursor.setPosition(10, 20);`).

```
# Fonction de vérification pour l'instruction setPosition
def check_set_position(self, node):
    cursor_name = node["cursor"] # Récupère le nom du curseur depuis le nœud

    # Vérifie si le curseur est déclaré
    if cursor_name not in self.cursor:
        self.errors.append(f"Cursor '{cursor_name}' is not declared.")
        return

    try:
        # Extraction et validation des coordonnées x et y
        x = self.extract_value(node["x"])
        y = self.extract_value(node["y"])

        # Vérifie que x et y sont des entiers
        if not isinstance(x, int) or not isinstance(y, int):
            self.errors.append(f"Position values for cursor '{cursor_name}' must be integers")
    except ValueError as e:
        # Ajout d'une erreur en cas de valeur incorrecte
        self.errors.append(str(e))
```

4.4 Analyse de la Fonction

- **Validation du Curseur** : La fonction commence par vérifier si le curseur mentionné dans le nœud (`node["cursor"]`) a été préalablement déclaré. Si ce n'est pas le cas, une erreur est ajoutée à la liste des erreurs.
- **Extraction des Coordonnées** : Les coordonnées `x` et `y` sont extraites du nœud via une fonction auxiliaire `extract_value`. Cette extraction s'assure que les valeurs associées sont conformes.
- **Validation des Types** : Une fois les coordonnées extraites, la fonction vérifie qu'elles sont de type entier. Toute anomalie entraîne une erreur signalée.
- **Gestion des Exceptions** : Si une erreur survient lors de l'extraction des valeurs (par exemple, un type inattendu), elle est capturée et ajoutée à la liste des erreurs.

4.5 Exemple d'AST et Résultat

Voici un exemple de nœud AST correspondant à une instruction de positionnement :

```
{
  "type": "SET_POSITION",
  "cursor": "mainCursor",
  "x": {"type": "VALUE", "value": 10},
  "y": {"type": "VALUE", "value": 20}
}
```

Traitement par l'Interpréteur :

- Si `mainCursor` a été déclaré et que les valeurs `x` et `y` sont des entiers, l'instruction est validée.
- Si `mainCursor` n'est pas déclaré, une erreur est ajoutée : `Cursor 'mainCursor' is not declared`.
- Si `x` ou `y` n'est pas un entier, une erreur similaire est ajoutée : `Position values for cursor 'mainCursor' must be integers`.

4.6 Importance de l'Interpréteur

Cette fonction illustre comment l'interpréteur garantit la cohérence et la validité des instructions. Ce processus de validation permet de prévenir les erreurs avant même la phase de compilation, améliorant ainsi la robustesse et la fiabilité du projet Draw++.

5 Le Compiler : Traduction en C

5.1 Le Compilateur de Draw++

L'objectif du compilateur est de traduire le code écrit en langage Draw++ en un programme exécutable, à travers une série d'étapes claires. Le compilateur prend comme entrée l'*Abstract Syntax Tree* (AST) généré par le parseur et produit du code en langage C. Ce code est ensuite compilé en un exécutable en utilisant un compilateur C standard.

5.2 Rôle du Compilateur dans le Projet Draw++

Le compilateur a plusieurs objectifs clés :

- **Génération du Code C** : Traduire les structures syntaxiques du langage Draw++ en instructions valides en C.
- **Utilisation de SDL2** : Intégrer des bibliothèques graphiques comme SDL2 pour permettre le rendu visuel des dessins.
- **Production d'un Exécutable** : Créer un fichier exécutable à partir du code C généré, afin que l'utilisateur puisse exécuter son programme de dessin sans interagir directement avec le code C.

5.3 Fonctionnement du Compilateur

Le fichier `compiler.py` contient les fonctions qui assurent le processus de compilation. Voici une vue d'ensemble de son fonctionnement :

- **Génération du Code C** : La fonction principale `generate_c_code` transforme l'AST fourni en code C structuré.
- **Sauvegarde du Code** : Le code C est écrit dans un fichier (par exemple `output.c`) grâce à la fonction `save_to_file`.
- **Compilation du Code C** : La fonction `compile_c_to_exe` appelle un compilateur C (comme GCC) pour produire un exécutable à partir du fichier `output.c`.

6 L’IDE de Draw++ : Aperçu et Fonctionnalités

6.1 Introduction

L’IDE (Éditeur de Développement Intégré) de Draw++ est un outil essentiel qui accompagne le langage de programmation Draw++. Il a été conçu pour simplifier l’écriture, le test et le débogage du code Draw++, tout en offrant une interface intuitive et accessible. Cette section présente en détail les fonctionnalités de cet IDE, son fonctionnement interne, et son interaction avec les différents composants comme le lexer, le parser, l’interpréteur et le compilateur.

6.2 Fonctionnalités principales

- **Interface utilisateur intuitive**

L’interface utilisateur de l’IDE, construite avec `tkinter`, est épurée et fonctionnelle. Elle inclut deux zones principales :

- **Zone d’édition du code** : Permet aux utilisateurs d’écrire et de modifier leur code en Draw++.
- **Zone d’affichage des erreurs** : Affiche les messages d’erreur, qu’ils soient syntaxiques, sémantiques ou issus de l’interprétation.

- **Gestion des fichiers**

L’IDE permet de créer, ouvrir, modifier et sauvegarder des fichiers Draw++ (extension `.draw++`). Ces opérations sont effectuées via les méthodes suivantes :

- `new_file` : Crée un nouveau fichier en vidant la zone de texte.
- `open_file` : Charge un fichier existant dans l’IDE.
- `save_file` : Sauvegarde le contenu édité.

- **Exécution et validation du code**

Deux fonctionnalités clés sont intégrées :

- **Exécution du code** : Traduit le code Draw++ en C, compile ce code et exécute l’exécutable final via un pipeline utilisant le lexer, le parser, l’interpréteur et le compilateur.
- **Validation du code** : Permet de vérifier la validité syntaxique et sémantique du code sans l’exécuter.

- **Interaction interactive**

Lorsque le code est exécuté, l’IDE offre un retour immédiat sur les graphiques générés via SDL2, permettant aux utilisateurs de visualiser directement l’impact de leur code.

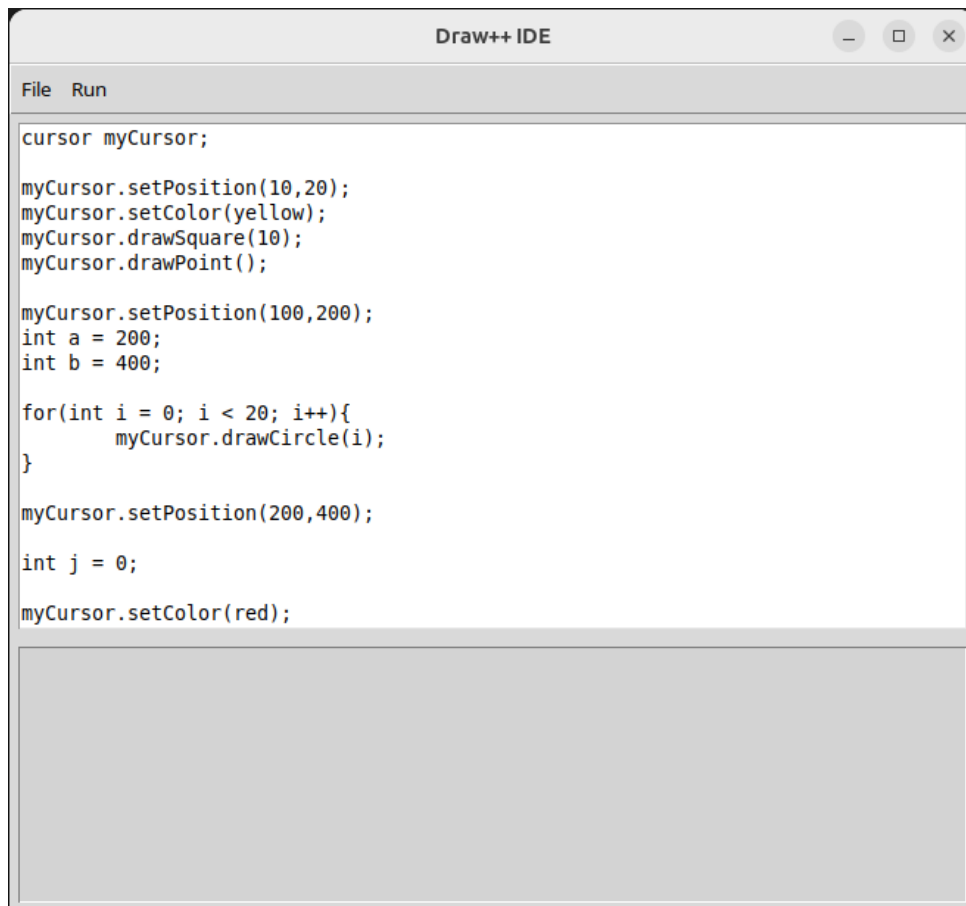


FIGURE 7 – Interface de l'IDE de DRAW ++

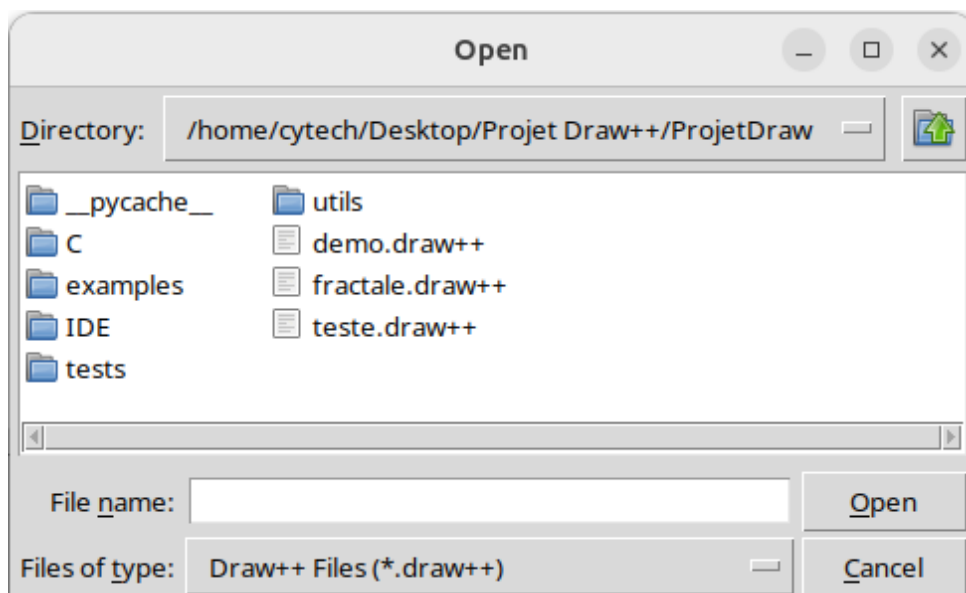
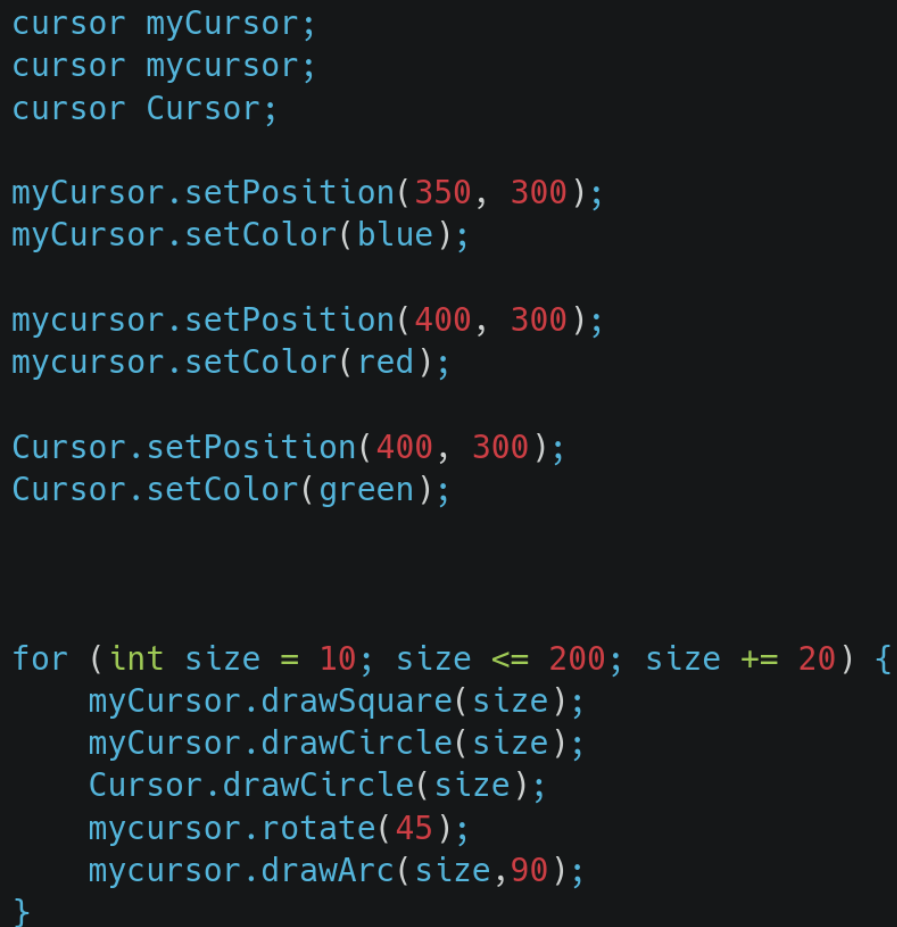


FIGURE 8 – Gestionnaire de fichier intégré

Exemples d'Exécution

Voici quelques exemples d'utilisation du langage **Draw++** et les rendus graphiques correspondants. Chaque exemple présente le code source et le résultat produit.

Exemple 1



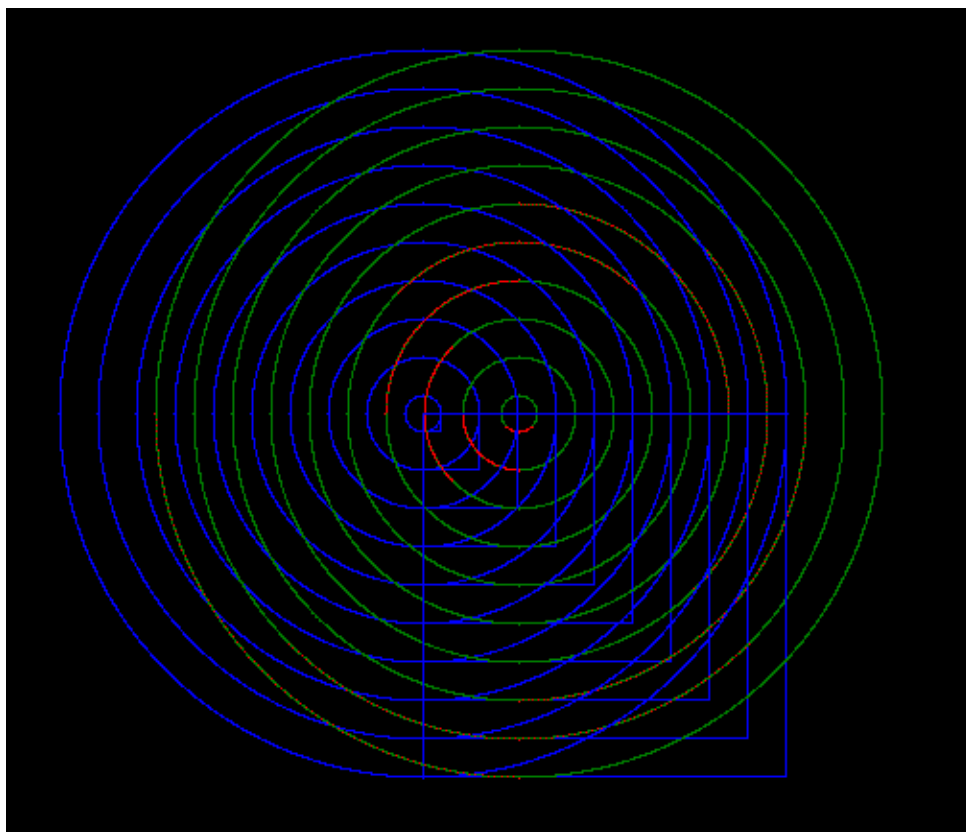
```
cursor myCursor;
cursor mycursor;
cursor Cursor;

myCursor.setPosition(350, 300);
myCursor.setColor(blue);


mycursor.setPosition(400, 300);
mycursor.setColor(red);

Cursor.setPosition(400, 300);
Cursor.setColor(green);

for (int size = 10; size <= 200; size += 20) {
    myCursor.drawSquare(size);
    myCursor.drawCircle(size);
    Cursor.drawCircle(size);
    mycursor.rotate(45);
    mycursor.drawArc(size, 90);
}
```



Exemple 2 :



```
cursor myCursor;
    cursor Cursor;
    int i = 0;
    int j = 50;
    int g = 100;
    int a = 25;
    int b = 50;
    while(i < 100){
        myCursor.setPosition(j,g);
        Cursor.setPosition(a,b);
        j += 5
        g += 2
        a += 3
        b += 8
        myCursor.setColor(red);
        Cursor.setColor(blue);

        myCursor.drawLine(10);

        myCursor.rotate(90);

        myCursor.drawPoint( );

        myCursor.drawArc(50, 90);

        myCursor.drawCircle(50);

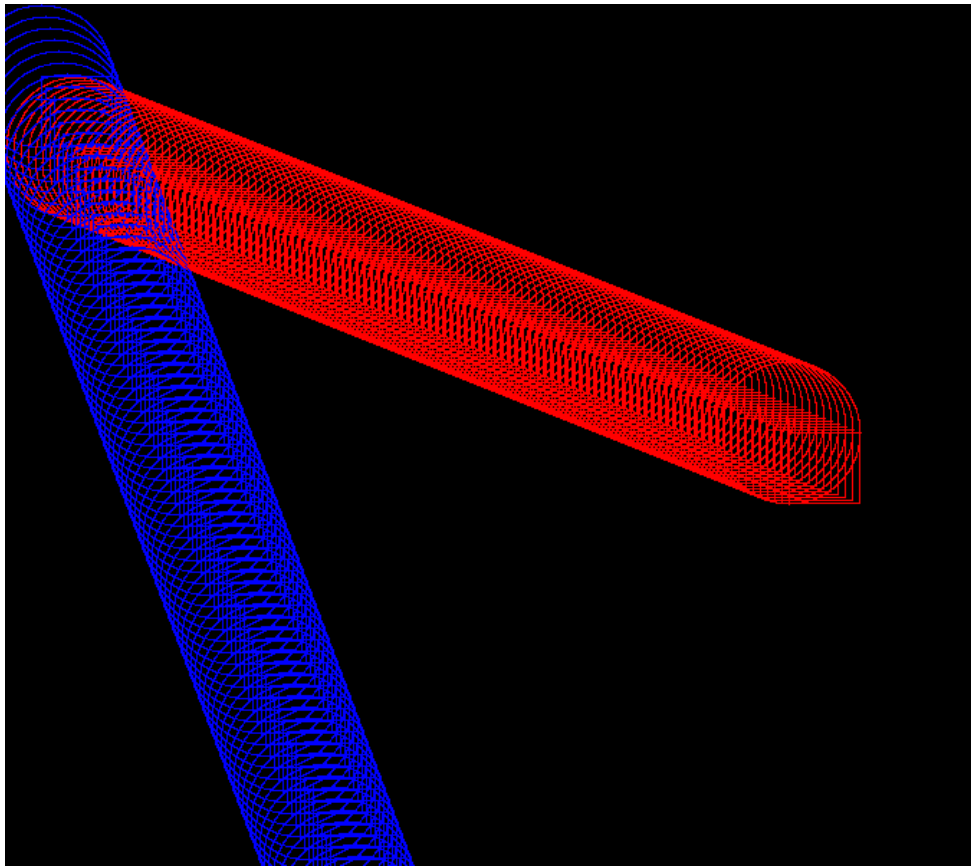
        myCursor.drawSquare(50);

        Cursor.drawLine(10);

        Cursor.rotate(90);

        Cursor.drawPoint( );

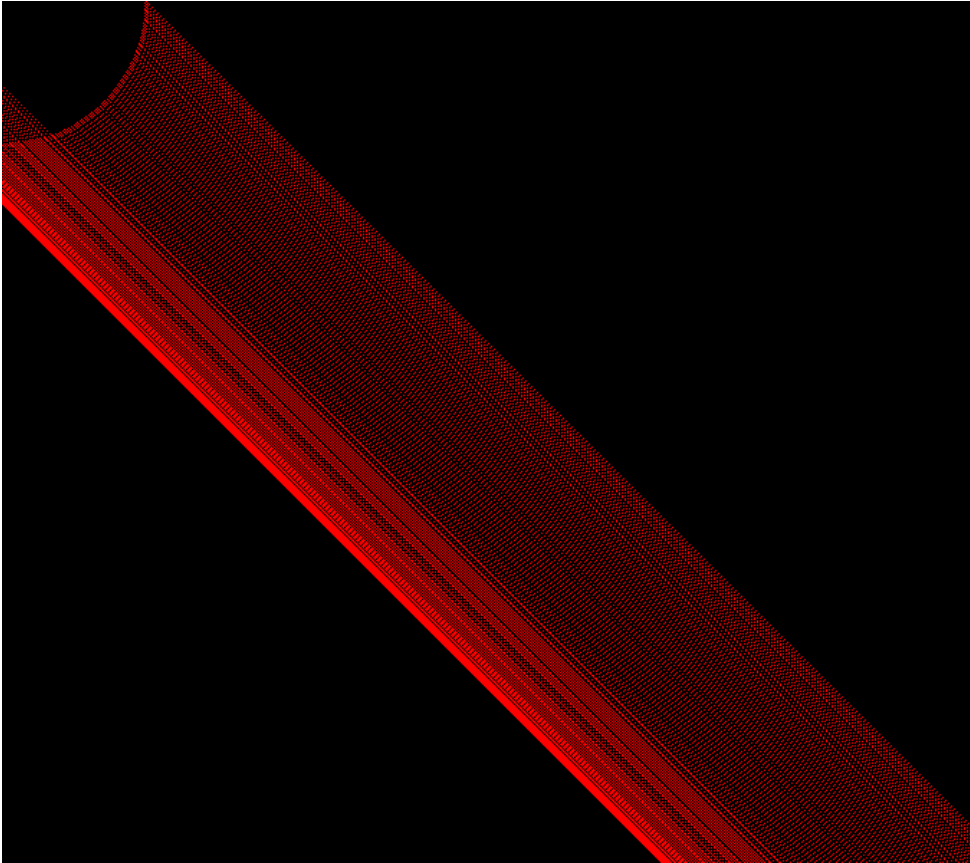
        Cursor.drawArc(50, 90);
```



Exemple 3 :



```
cursor myCursor;  
  
int a = 0;  
int b = 0;  
  
myCursor.setPosition(1,1);  
  
int j = 0;  
  
myCursor.setColor(red);  
  
while(j < 10000){  
    myCursor.drawArc(100,200);  
    myCursor.setPosition(a,b);  
    a+=2  
    b+=2  
    j+=2  
}
```

7 Conclusion

Le projet Draw++ a permis de concevoir un langage de programmation dédié à la création graphique, associé à un environnement de développement intégré (IDE). Ce langage, avec sa syntaxe intuitive, offre une expérience accessible pour dessiner des formes et créer des illustrations graphiques.

L'intégration de composants comme le lexer, le parser, l'interpréteur, et le compilateur garantit une analyse précise et une exécution fiable du code. L'IDE complète cette architecture en offrant un espace interactif pour écrire, tester et visualiser les résultats.

Ce projet a été une opportunité d'explorer les concepts clés des langages de programmation afin d'enrichir ses compétences.