Rapport Projet Draw++

GIA1 - Groupe 2
Adam Terrak, Deulyne Destin, Mathias Vinkovic, Louaye Saghir, Alice Franco

January 12, 2025

Table of Contents

1 Introduction

Le projet Draw++ s'inscrit dans le domaine de la conception de langages spécifiques, souvent appelés *DSL* (Domain Specific Languages). Ce langage a été développé pour simplifier la création et l'animation de formes géométriques tout en offrant un ensemble d'instructions claires et faciles à apprendre.

Draw++ se distingue par une syntaxe inspirée de langages populaires tels que Python et C, tout en se concentrant sur la manipulation graphique. Les utilisateurs peuvent non seulement dessiner des formes (cercles, rectangles, lignes, etc.), mais également contrôler leur animation, leur couleur, et leur position sur un écran virtuel. En intégrant des constructions comme les conditions et les boucles, Draw++ permet aussi de créer des séquences complexes et dynamiques.

Pour rendre ce langage fonctionnel, nous avons développé un traducteur capable d'interpréter et de vérifier le code écrit en Draw++. Ce traducteur repose sur deux éléments essentiels :

- Un **analyseur lexical** (*lexer*), qui identifie les composants élémentaires du langage : mots-clés, identifiants, opérateurs, etc.
- Un **analyseur syntaxique** (*parser*), qui vérifie que le code respecte les règles grammaticales du langage et construit une structure logique appelée *arbre syntaxique abstrait* (AST).

Nous avons utilisé la bibliothèque open-source **PLY** (*Python Lex-Yacc*) pour la mise en œuvre du lexer et du parser. Cette bibliothèque nous a permis de définir une grammaire complète pour Draw++ et de détecter efficacement les erreurs lexicales, syntaxiques et sémantiques dans les programmes.

Ce rapport est structuré comme suit :

- Une description complète de la syntaxe et des instructions de Draw++ ;
- Les détails de conception et de fonctionnement du traducteur ;
- La grammaire formelle du langage ;
- Une analyse des erreurs possibles et de leur gestion ;
- Des exemples d'exécutions accompagnés d'illustrations graphiques ;
- Une conclusion résumant les résultats obtenus et les perspectives d'amélioration.

Le langage Draw++ vise à rendre accessible la création graphique et l'animation à un large public, tout en constituant un exemple concret d'application de concepts fondamentaux en informatique, tels que la conception de langages et l'analyse syntaxique.

—

2 La syntaxe de Draw++

Le projet Draw++ consiste à concevoir un langage spécifique permettant de dessiner des formes géométriques, d'animer des objets et de manipuler des curseurs sur un écran. Le langage est divisé en instructions élémentaires (dessin, déplacement, couleur) et évoluées (conditions, boucles, regroupement d'instructions).

Pour traiter ce langage, nous avons conçu un traducteur composé d'un analyseur lexical (lexer) et d'un analyseur syntaxique (parser) à l'aide de la bibliothèque open-source **PLY**. Ce rapport explique en détail la syntaxe, le fonctionnement du lexer et du parser, la grammaire du langage et les exemples d'exécution. Draw++ suit une syntaxe simple inspirée de langages tels que Python et C, avec des instructions structurées.

2.1 Instructions élémentaires

Les instructions élémentaires définissent les fonctionnalités de base du langage **Draw++**. Elles permettent aux utilisateurs d'interagir facilement avec les curseurs et de dessiner des formes simples. Voici une présentation des instructions élémentaires, leur syntaxe et leurs arguments :

2.1.1 Formes supportées

Le langage **Draw++** permet de dessiner différentes formes géométriques grâce à l'instruction `draw`. Chaque forme a une syntaxe spécifique et des arguments associés.

- **Cercle (`circle`) :**
  - **Syntaxe :**

```
draw circle(mode, remplissage, couleur, centerX, centerY,
   radius)
```

  - **Arguments :**
    - `mode` : `instant` ou `animated`.
    - `remplissage` : `filled` ou `empty`.
    - `couleur` : `red`, `green`, `blue`, etc.
    - `centerX, centerY` : Coordonnées du centre.
    - `radius` : Rayon du cercle.
  - **Exemple :**

```
(instant, filled, red, 100, 100, 50)
```

- **Rectangle** (`rectangle`) :
  - **Syntaxe :**

```
draw rectangle(mode, remplissage, couleur, x, y, width,
   height)
```

  - **Arguments :**
    - `mode`, `remplissage`, et `couleur` : Même signification que pour le cercle.
    - `x, y` : Coordonnées du coin supérieur gauche.
    - `width, height` : Largeur et hauteur du rectangle.
  - **Exemple :**

```
draw rectangle(animated, empty, blue, 50, 50, 150, 100)
```

- **Carré** (`square`) :
  - **Syntaxe :**

```
draw square(mode, remplissage, couleur, x, y, sideLength)
```

  - **Arguments :**
    - `x, y` : Coordonnées du coin supérieur gauche.
    - `sideLength` : Longueur d'un côté.
  - **Exemple :**

```
draw square(instant, filled, green, 200, 200, 75)
```

- **Triangle** (`triangle`) :
  - **Syntaxe :**

```
draw triangle(mode, remplissage, couleur, centerX, centerY,
   radius)
```

  - **Arguments :**
    - `centerX, centerY` : Coordonnées du centre du triangle.
    - `radius` : Taille du triangle du sommet au centre.
  - **Exemple :**

```
draw triangle(animated, filled, yellow, 150, 150, 50)
```

- **Ligne (`line`) :**
  - **Syntaxe :**

```
draw line(mode, couleur, x1, y1, x2, y2, thickness)
```

  - **Arguments :**
    - `x1, y1` : Coordonnées du point de départ.
    - `x2, y2` : Coordonnées du point d'arrivée.
    - `thickness` : Épaisseur de la ligne.
  - **Exemple :**

```
draw line(instant, red, 10, 10, 200, 200, 5)
```

- **Ellipse (`ellipse`) :**
  - **Syntaxe :**

```
draw ellipse(mode, remplissage, couleur, centerX, centerY,
   radiusX, radiusY)
```

  - **Arguments :**
    - `radiusX, radiusY` : Rayons horizontal et vertical de l'ellipse.
  - **Exemple :**

```
draw ellipse(animated, empty, purple, 100, 100, 80, 50)
```

- **Polygone (`polygon`) :**
  - **Syntaxe :**

```
draw polygon(mode, remplissage, couleur, centerX, centerY,
   radius, sides)
```

  - **Arguments :**
    - `radius` : Rayon du polygone.
    - `sides` : Nombre de côtés du polygone.
  - **Exemple :**

```
draw polygon(instant, filled, cyan, 300, 300, 50, 5)
```

- **Arc (`arc`) :**
  - **Syntaxe :**

```
draw arc(mode, couleur, centerX, centerY, radius,
   startAngle, endAngle)
```

  - **Arguments :**
    - `startAngle, endAngle` : Angles de départ et de fin en degrés.
  - **Exemple :**

```
draw arc(animated, blue, 400, 400, 100, 0, 180)
```

2.1.2 Autres instructions

En complément des instructions de dessin, **Draw++** propose des commandes permettant de configurer des propriétés globales et des paramètres de la fenêtre de dessin. Voici une description détaillée de ces instructions :

- **Définir la taille du curseur (`cursor size`) :**
  - **Syntaxe :**

```
set cursor size(size)
```

  - **Arguments :**
    - `size` : Taille du curseur en pixels (valeur entière positive).
  - **Exemple :**

```
set cursor size(10)
```

- **Définir la couleur du curseur (`cursor color`) :**
  - **Syntaxe :**

```
set cursor color(color)
```

  - **Arguments :**
    - `color` : Couleur du curseur (ex. `red`, `green`, `blue`, etc.).
  - **Exemple :**

```
set cursor color(blue)
```

- **Définir la taille de la fenêtre (`window size`) :**
  - **Syntaxe :**

```
set window size(width, height)
```

  - **Arguments :**
    - `width` : Largeur de la fenêtre en pixels (valeur entière positive).
    - `height` : Hauteur de la fenêtre en pixels (valeur entière positive).
  - **Exemple :**

```
set window size(800, 600)
```

- **Définir la couleur de la fenêtre (`window color`) :**
  - **Syntaxe :**

```
set window color(color)
```

  - **Arguments :**
    - `color` : Couleur d'arrière-plan de la fenêtre (ex. `white`, `black`, `yellow`, etc.).
  - **Exemple :**

```
set window color(green)
```

2.2 Instructions évoluées

Les instructions évoluées permettent d'intégrer des fonctionnalités avancées dans le langage **Draw++**. Elles incluent les conditions, les boucles et les animations.

- **Conditions :**
  Permettent d'exécuter des instructions en fonction d'une condition logique. Voici la syntaxe générale :

```
if (condition) {
    // instructions si la condition est vraie
} elif (autreCondition) {
    // instructions si l'autre␣condition␣est␣vraie
}␣else␣{
␣␣␣␣//␣instructions␣sinon
}
```

Listing 1: Syntaxe des conditions

Exemple d'utilisation :

```
if (x < 10) {
    (instant, filled, red, 50, 50, 25)
} elif (x < 20) {
    draw rectangle(animated, filled, green, 10, 20, 100, 50)
} else {
    draw line(instant, blue, 0, 0, 200, 200, 5)
}
```

Listing 2: Exemple d'utilisation des conditions

- **Boucles :**
  Les boucles permettent de répéter des instructions. Voici les différentes boucles supportées dans **Draw++** :

  – `while` : Répète tant qu'une condition est vraie.

  ```
  while (condition) {
      //instructions repetees
  }
  ```

  Listing 3: Boucle while

  – `for` : Répète un nombre défini de fois avec une initialisation, une condition et une incrémentation.

  ```
  for (var i = 0; i < 10; i = i + 1) {
      draw circle(instant, filled, red, 50, 50, 25)
  }
  ```

  Listing 4: Boucle for

  – `do-while` : Exécute les instructions au moins une fois, puis répète tant qu'une condition est vraie.

  ```
  do {
      move(10, 20)
  } while (x < 100)
  ```

  Listing 5: Boucle do-while

- **Animations :**
  Permettent d'animer des formes. Voici un exemple d'utilisation :

```
animate((animated, filled, red, 50, 50, 25))
```

Listing 6: Exemple d'animation

- **Groupement d'instructions :**
  Les blocs regroupent plusieurs instructions pour structurer le code. Voici la syntaxe :

```
{
    // groupe d'instructions
}
```

Listing 7: Bloc d'instructions

Exemple d'utilisation :

```
{
    color(red)
    move(50, 50)
    (instant, filled, red, 50, 50, 25)
}
```

Listing 8: Exemple de bloc d'instructions

—

3 Fonctionnement du traducteur

Le traducteur est composé de deux parties principales : un analyseur lexical (**lexer**) et un analyseur syntaxique (**parser**). Ces deux composants travaillent en coordination pour interpréter le code source et produire un code intermédiaire.

3.1 Présentation générale de PLY

Pour la construction du traducteur, nous avons utilisé la bibliothèque open source **PLY** (*Python Lex-Yacc*). PLY est une implémentation de Lex et Yacc pour Python, conçue pour faciliter la création d'analyseurs lexicaux et syntaxiques.

Caractéristiques principales de PLY :

- **Basé sur Lex et Yacc :** PLY offre une syntaxe et un fonctionnement similaires à ces outils classiques utilisés pour le traitement de langages.
- **Complète et robuste :** Elle prend en charge des grammaires complexes avec des fonctionnalités comme la gestion des conflits et des ambiguïtés.
- **Écrite en Python :** Permet une intégration fluide dans des projets Python avec une simplicité d'utilisation et une syntaxe intuitive.
- **Gestion des erreurs :** PLY inclut des mécanismes pour signaler et gérer les erreurs lexicales et syntaxiques.

Modules principaux de PLY : PLY se compose de deux modules :

- **Lex :** Utilisé pour construire un analyseur lexical en associant des expressions régulières à des actions spécifiques.
- **Yacc :** Utilisé pour construire un analyseur syntaxique basé sur des règles de production. Ces règles décrivent comment assembler les tokens en structures syntaxiques valides.

Dans le contexte de notre projet Draw++, PLY a été utilisé pour construire le lexer et le parser, composants essentiels pour interpréter et vérifier le code source.

3.2 Analyseur lexical (Lexer)

L'analyseur lexical, ou **lexer**, est la première étape du processus de traduction du code source. Son rôle est de lire le code ligne par ligne, caractère par caractère, et de le découper en une série de **tokens**. Un token représente une unité significative du langage, telle qu'un mot-clé (`draw`, `if`), un identifiant (`variable_name`), un opérateur (`+`, `-`) ou encore un délimiteur (`(`, `)`).

Utilisation des expressions régulières (regex) : Pour construire le lexer, nous avons utilisé des **expressions régulières** (*regex*). Les regex permettent de décrire précisément les motifs que chaque token peut correspondre, comme des mots-clés, des identifiants ou des nombres. Chaque règle lexicale est définie en associant une regex à un type de token ou une fonction spécifique.

Fonctionnement du lexer : Le lexer parcourt le code source de manière séquentielle et applique les regex définies pour identifier les tokens. Lorsqu'une correspondance est trouvée, un token est généré avec les informations suivantes :

- **Le type du token :** Indique sa catégorie, comme `IDENTIFIER` (identifiant) ou `NUMBER` (nombre).
- **La valeur du token :** Le contenu correspondant dans le code source (par exemple, le nom d'une variable ou une valeur numérique).
- **La position :** La ligne et la colonne où le token a été trouvé, utile pour le débogage et la gestion des erreurs.

```
# Definition des types de tokens
tokens = ['IDENTIFIER', 'NUMBER', 'PLUS', 'MINUS']

# Regles pour les operateurs
t_PLUS = r'\+'       # Correspond au caractere '+'
t_MINUS = r'-'        # Correspond au caractere '-'

# Regle pour les identifiants (variables ou mots-cles)
t_IDENTIFIER = r'[a-zA-Z_][a-zA-Z0-9_]*'  # Une lettre ou '_',
   suivie de lettres/chiffres

# Regle pour les nombres
def t_NUMBER(t):
    r'\d+'              # Une ou plusieurs chiffres consecutifs
    t.value = int(t.value)  # Conversion en entier
    return t
```

Listing 9: Exemple de regles lexicales avec regex

Processus de transformation : Lorsque le lexer rencontre un code comme celui-ci :

```
draw circle(instant, filled, red, 100, 100, 50)
```

Les regex détectent les motifs correspondants et le lexer génère la liste de tokens suivante :

```
[
    ('IDENTIFIER', 'draw'),
    ('IDENTIFIER', 'circle'),
    ('LPAREN', '('),
    ('INSTANT', 'instant'),
    ('COMMA', ','),
    ('FILLED', 'filled'),
    ('COMMA', ','),
    ('RED', 'red'),
    ('COMMA', ','),
    ('NUMBER', 100),
    ('COMMA', ','),
    ('NUMBER', 100),
    ('COMMA', ','),
    ('NUMBER', 50),
    ('RPAREN', ')')
]
```

Listing 10: Exemple de tokens générés

Chaque token contient son type et sa valeur, ce qui permet à l'analyseur syntaxique (parser) de reconstruire la structure logique du programme.

Gestion des erreurs lexicales : Le lexer est également responsable de détecter les erreurs lexicales, comme des caractères invalides ou des tokens mal formés. En cas d'erreur, le lexer lève une exception ou affiche un message informatif :

```
def t_error(t):
    print(f"LexicalError: Illegal character '{t.value[0]}' at line
       {t.lineno}, column {t.lexpos}")
    t.lexer.skip(1)  # Ignore le caractere invalide et continue
```

Listing 11: Gestion des erreurs lexicales

Conclusion : L'analyseur lexical joue un rôle crucial dans le traducteur. Grâce à l'utilisation des regex, nous avons pu définir un ensemble de règles précises et modulaires pour détecter les différents éléments de syntaxe du langage Draw++. Cela garantit que le code source est transformé en une structure exploitable pour les étapes suivantes, tout en identifiant et signalant les erreurs lexicales de manière efficace.

3.3 Analyseur syntaxique (Parser)

L'analyseur syntaxique, ou **parser**, est une étape clé dans le fonctionnement du traducteur. Il reçoit les tokens produits par le lexer et les organise en une structure hiérarchique appelée **arbre syntaxique abstrait** (AST). Cet arbre représente la structure logique du programme et sert de base pour les étapes ultérieures, comme la génération de code intermédiaire.

Rôle principal du parser :

- **Organisation logique :** Le parser assemble les tokens selon les règles de grammaire définies pour le langage Draw++.
- **Vérification syntaxique :** Il s'assure que le code respecte la grammaire, détecte les erreurs syntaxiques et les signale.
- **Production d'un AST :** Il construit un arbre syntaxique abstrait qui reflète la logique du programme.

Construction récursive : Le parser utilise des règles récursives pour construire l'AST. Par exemple, une règle pour un programme peut être définie comme suit :

```
def p_programme(p):
    '''programme : instruction
                 | programme instruction'''
    p[0] = [p[1]] if len(p) == 2 else p[1] + [p[2]]
```

Listing 12: Règle pour un programme

Cette règle indique que :

- Un programme peut être composé d'une seule instruction (`instruction`).
- Un programme peut également être une suite d'instructions (`programme instruction`).
- Le parser traite récursivement chaque instruction et les assemble dans une liste.

Validation syntaxique : Pour chaque construction syntaxique du langage, une règle de production est définie dans le parser. Prenons l'exemple de l'instruction `draw` :

```
def p_draw(p):
    '''draw : DRAW forme LPAREN parametres RPAREN'''
    p[0] = ('draw', p[2], p[4], p.lineno(1))
```

Listing 13: Règle pour l'instruction draw

Cette règle :

- Reconnaît une instruction `draw` suivie d'une `forme`, d'une liste de `parametres` encadrée par des parenthèses (`LPAREN` et `RPAREN`).
- Retourne un nœud dans l'AST contenant :
  - Le type de l'instruction (`'draw'`).
  - La forme (`p[2]`).
  - Les paramètres (`p[4]`).
  - La position dans le code (`p.lineno(1)`) pour faciliter le débogage.

Détection et gestion des erreurs syntaxiques : Le parser est conçu pour détecter les erreurs syntaxiques et les signaler de manière informative. Une fonction spéciale, `p_error`, est appelée lorsqu'un token inattendu ou une structure invalide est rencontrée :

```
def p_error(p):
    if p:
        print(f"SyntaxError: Unexpected '{p.value}' at line
           {p.lineno}")
    else:
        print("SyntaxError: Unexpected end of input")
```

Listing 14: Gestion des erreurs syntaxiques

- `p.value` fournit le token problématique.
- `p.lineno` indique la ligne où l'erreur s'est produite.
- Si le programme se termine de manière inattendue, un message indique une fin d'entrée incorrecte.

Exemple de règles avancées : Prenons l'exemple des boucles `while` et `do-while` dans le langage Draw++ :

```
def p_boucle(p):
    '''boucle : WHILE LPAREN expression RPAREN bloc
              | DO bloc WHILE LPAREN expression RPAREN'''
    if p[1] == 'while':
        p[0] = ('while', p[3], p[5])
    else:
        p[0] = ('do-while', p[2], p[5])
```

Listing 15: Règle syntaxique pour les boucles

Cette règle :

- Gère deux types de boucles :
  - La boucle `while`, qui exécute un bloc tant qu'une condition (`expression`) est vraie.
  - La boucle `do-while`, qui exécute un bloc au moins une fois avant de vérifier la condition.
- Ajoute un nœud correspondant dans l'AST avec les éléments :
  - Le type de boucle (`'while'` ou `'do-while'`).
  - La condition (`p[3]` pour `while` ou `p[5]` pour `do-while`).
  - Le bloc d'instructions (`p[5]` pour `while` ou `p[2]` pour `do-while`).

Production de l'AST : À partir de règles comme celles ci-dessus, le parser produit un AST structuré. Voici un exemple d'AST généré pour un programme simple :

```
[
    ('draw', 'circle', ['filled', 'red', 100, 100, 50], 1),
    ('while', ('<', 'x', 10), [
        ('draw', 'rectangle', ['empty', 'blue', 50, 50, 150, 100],
           3)
    ])
]
```

Listing 16: Exemple d'AST

- Chaque nœud correspond à une instruction ou une structure syntaxique.
- Les sous-nœuds représentent les composants internes, comme les paramètres ou les blocs.

Conclusion : L'analyseur syntaxique est une étape fondamentale dans le processus de traduction. Il transforme le code source en une structure logique bien définie tout en vérifiant la conformité syntaxique. En cas d'erreur, il fournit des messages explicites pour faciliter le débogage. Grâce à sa conception modulaire et aux règles définies, le parser peut facilement être étendu pour prendre en charge de nouvelles constructions syntaxiques dans le langage Draw++.

—

4 Grammaire de Draw++

La grammaire complète de Draw++ est décrite ci-dessous.
Elle inclut toutes les structures syntaxiques supportées par le langage, conformément au lexer fourni.

```
<programme> ::= <instruction> | <programme> <instruction>

<instruction> ::= <dessin> | <deplacement> | <rotation> | <couleur>
                | <assignation> | <declaration> | <animation>
                | <conditionnelle> | <boucle> | "{" <programme> "}"

<dessin> ::= "draw" <forme> "(" <parametres> ")"
<forme> ::= "line" | "circle" | "square" | "rectangle" | "triangle"
   | "ellipse" | "polygon"

<parametres> ::= <expression> ("," <expression>)*

<deplacement> ::= "move" "(" <expression> "," <expression> ")"
<rotation> ::= "rotate" "(" <expression> ")"
<couleur> ::= "color" "(" <nom_couleur> ")"

<assignation> ::= <identifiant> "=" <expression>
<declaration> ::= "var" <identifiant> "=" <expression>

<animation> ::= "animate" "(" <instruction> ")"

<conditionnelle> ::= "if" "(" <expression_logique> ")" <instruction>
                   | "if" "(" <expression_logique> ")"
                      <instruction> "else" <instruction>

<boucle> ::= "while" "(" <expression_logique> ")" <instruction>
           | "do" <instruction> "while" "(" <expression_logique> ")"
           | "for" "(" <assignation> ";" <expression_logique> ";"
              <modification ")" <instruction>

<modification> ::= <identifiant> "=" <expression>

<expression_logique> ::= <expression> <comparateur> <expression>
```

```
<comparateur> ::= "==" | "!=" | "<" | ">" | "<=" | ">="

<expression> ::= <nombre>
               | <identifiant>
               | <expression> "+" <expression>
               | <expression> "-" <expression>
               | <expression> "*" <expression>
               | <expression> "/" <expression>
               | "(" <expression> ")"

<nombre> ::= une constante num rique (par ex. 10, 3.14)
<identifiant> ::= un nom valide de variable (par ex. x, y,
   maVariable)

<nom_couleur> ::= "red" | "blue" | "green" | "yellow" | "black" |
   "white"
                | "purple" | "cyan" | "orange"

<bloc> ::= "{" <instruction_list> "}"
<instruction_list> ::= <instruction> | <instruction_list>
   <instruction>
```

—

4.1 Éléments spécifiques ajoutés

Voici les éléments spécifiques ajoutés pour compléter la grammaire :

- **Formes supplémentaires :** - En plus des formes classiques comme `circle` et `rectangle`, des formes comme `ellipse` et `polygon` sont également supportées. - Les paramètres pour ces formes doivent être fournis sous forme de liste séparée par des virgules.
- **Couleurs étendues :** - Le langage supporte une palette de couleurs élargie, incluant `purple`, `cyan`, et `orange`, en plus des couleurs de base.
- **Boucles :** - Les boucles `for`, `while`, et `do-while` sont toutes incluses avec leurs variantes syntaxiques, comme les modifications incrémentales.
- **Blocs d'instructions :** - Les blocs permettent de regrouper plusieurs instructions sous forme de {}. Ils sont utilisés pour les boucles, les conditions, et les programmes imbriqués.

—

4.2 Exemple de programme Draw++

Voici un programme complet illustrant les différentes fonctionnalités de Draw++ :

```
var x = 1
var y = 10
set cursor color(blue)
set window size(500, 500)

while (x < 100) {
    draw circle(instant, filled, red, x, y, 50)
    x = x + 20
    if (x == 60) {
        draw ellipse(empty, blue, x, y, 100, 50)
    }
}

for (var i = 0; i < 5; i = i + 1) {
    draw rectangle(filled, green, x, y, 100, 50)
}
```

—

5 Analyse des erreurs

Les erreurs dans un programme Draw++ peuvent être détectées et signalées à différents niveaux : lexical, syntaxique, et sémantique.
De plus, le traducteur inclut un mécanisme de **suggestions** pour guider l'utilisateur dans la correction de ces erreurs.

5.1 Erreurs lexicales

Les erreurs lexicales surviennent lorsque le lexer rencontre des caractères ou des mots qui ne correspondent à aucun des tokens définis dans le langage.

Exemples d'erreurs lexicales :

- Un caractère inconnu, comme `$`, dans une ligne de code.
- Un identifiant mal formé qui ne respecte pas les règles définies pour les noms de variables (`var2@` par exemple).

Gestion des erreurs lexicales avec suggestions : Lorsque le lexer détecte une erreur, il lève un message explicite contenant la ligne, la colonne, et le caractère problématique, accompagné d'une suggestion. Par exemple :

```
LexicalError: Illegal character '$' at line 3, column 15.
Suggestion: Suggestion: Remove the '$' character or replace it with a valid one.
```

Voici la fonction utilisée pour gérer ces erreurs et fournir des suggestions :

```
def t_error(t):
    error_msg = f"Lexical error: Illegal character '{t.value[0]}'
       at line {t.lineno}, column {find_column(t.lexer.lexdata,
       t.lexpos)}"
    raise SyntaxError(error_msg)
```

Listing 17: Gestion des erreurs lexicales avec suggestions

5.2 Erreurs syntaxiques

Les erreurs syntaxiques apparaissent lorsque le code source ne respecte pas la structure définie par la grammaire du langage. Cela inclut :

- Parenthèses ou accolades manquantes (`draw circle (instant, filled , red , 50 , 50 , 25`).
- Ordre incorrect des paramètres (`move 100 , (50)`).
- Utilisation d'une structure non définie dans la grammaire.

Gestion des erreurs syntaxiques avec suggestions : Le parser détecte les erreurs de structure en suivant les règles de la grammaire. En cas d'erreur, il génère un message explicite, accompagné d'une suggestion pour corriger le problème. Par exemple :

```
SyntaxError: Unexpected ')' at line 5.
Suggestion: Check for missing opening '(' or mismatched parameters.
```

Voici la fonction gérant ces erreurs :

```
def p_error(p):
    if p:
        print(f"SyntaxError: Unexpected '{p.value}' at line
           {p.lineno}")
        print("Suggestion: Verify the structure of your code around
           this line.")
    else:
        print("SyntaxError: Unexpected end of input")
        print("Suggestion: Ensure all blocks and statements are
           properly closed.")
```

Listing 18: Gestion des erreurs syntaxiques avec suggestions

5.3 Erreurs sémantiques

Les erreurs sémantiques surviennent lorsque le code utilise des valeurs ou des types incorrects, bien que la structure soit syntaxiquement valide. Cela inclut :

- Passer un nombre au lieu d'une chaîne à `color (100)` au lieu de `color (red)`.
- Référencer une variable non initialisée (`draw circle (instant, filled, red, x, 50, 25)` alors que `x` n'a pas été déclarée).
- Utiliser une opération arithmétique sur des types incompatibles (`1 0 + " text "`).

Gestion des erreurs sémantiques avec suggestions : Les erreurs sémantiques sont détectées lors de l'analyse de l'AST. Le système fournit des suggestions pour corriger les erreurs détectées. Par exemple :

```
SemanticError: Invalid parameter type for 'color': expected string, got number at line 3
Suggestion: Use a valid color name like 'red' or 'blue'.

SemanticError: Variable 'x' not initialized at line 5.
Suggestion: Declare the variable 'x' before using it.
```

5.4 Résumé des types d'erreurs détectées et des suggestions

Type d'erreur	Description et exemples	Suggestions fournies
Lexical	Caractères inconnus ou identifiants mal formés. Exemple : `#`, `var2@`.	Vérifiez les caractères non valides et assurez-vous que les identifiants respectent les règles.
Syntaxique	Problèmes de structure, comme des parenthèses ou accolades manquantes. Exemple : `draw circle (instant, filled, red, 100 , 50 , 25`.	Vérifiez la structure des blocs et la présence de toutes les parenthèses/accolades.
Sémantique	Types incorrects ou références à des variables non initialisées. Exemple : `color (100)` ou `x + " text "`.	Utilisez des types corrects (ex. chaîne pour `color`) et déclarez toutes les variables avant de les utiliser.

Table 1: Résumé des types d'erreurs détectées dans Draw++ et des suggestions fournies

—

6 Exemples d'exécutions

6.1 Exemple d'un programme simple :

```
var x = 0
var y = 10

while (x < 100) {
    draw circle(instant, filled, red, x, y, 50)
    x = x + 20
    if (x == 60) {
        color(blue)
        draw ellipse(empty, blue, x, y, 100, 50)
    }
}

for (var i = 0; i < 5; i = i + 1) {
    draw rectangle(filled, green, x, y, 100, 50)
    move(i * 10, y + 20)
}
```

Listing 19: Programme Draw++

6.2 Résultats graphiques :

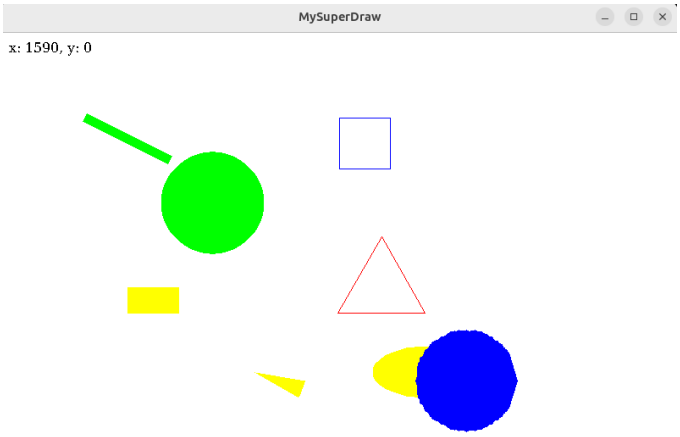

Figure 1: Affichage initial.

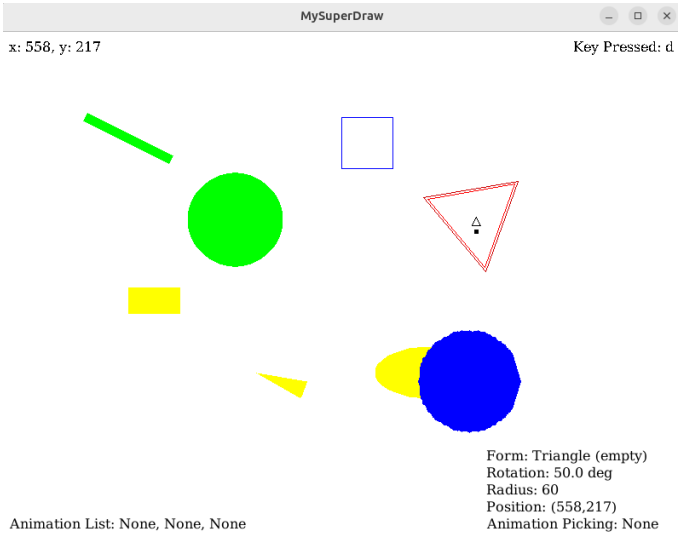

Figure 2: Rotation du triangle avec la touche d ou q.

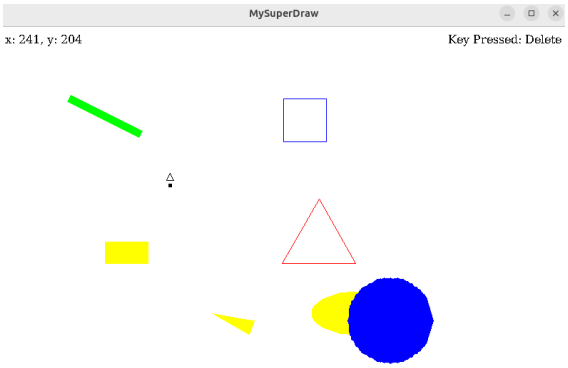

Figure 3: Suppression du cercle vert.

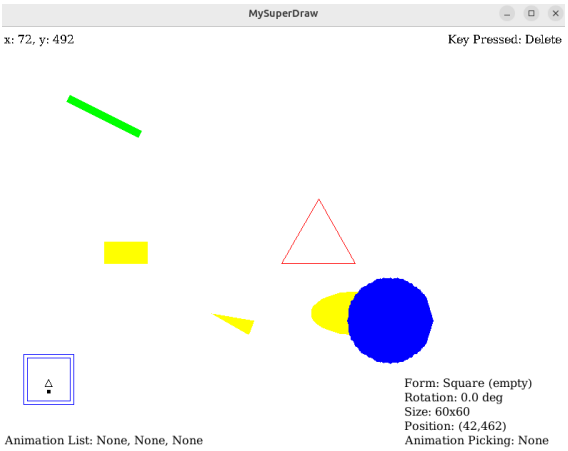

Figure 4: Déplacement du carré bleu.

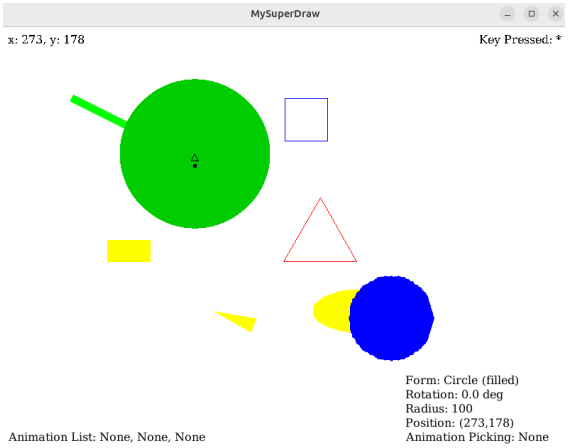

Figure 5: Zoom du cercle vert avec la touche *.

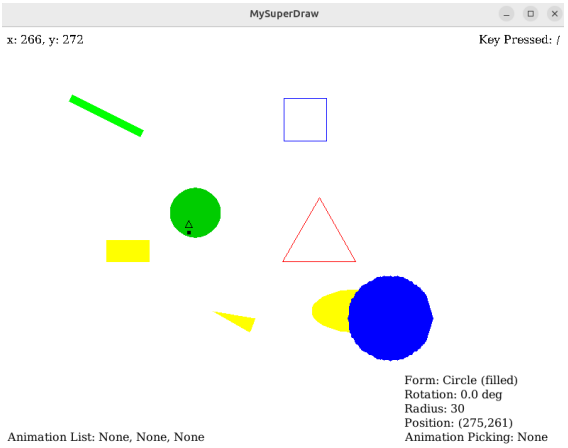

Figure 6: Dézoom du cercle vert avec la touche /.

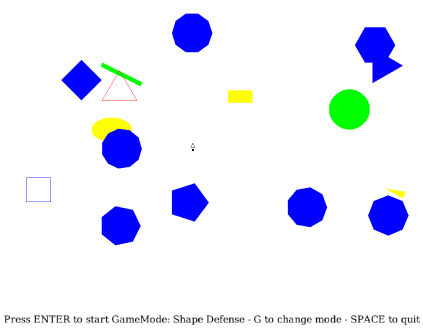

Figure 7: Game mode Shape Défense.

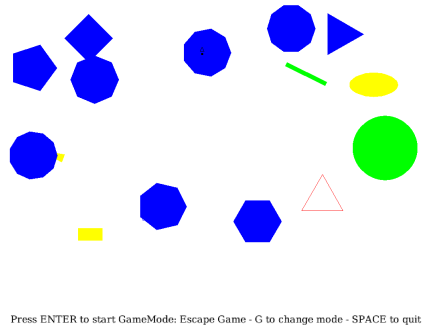

Figure 8: Game mode Escape Game.

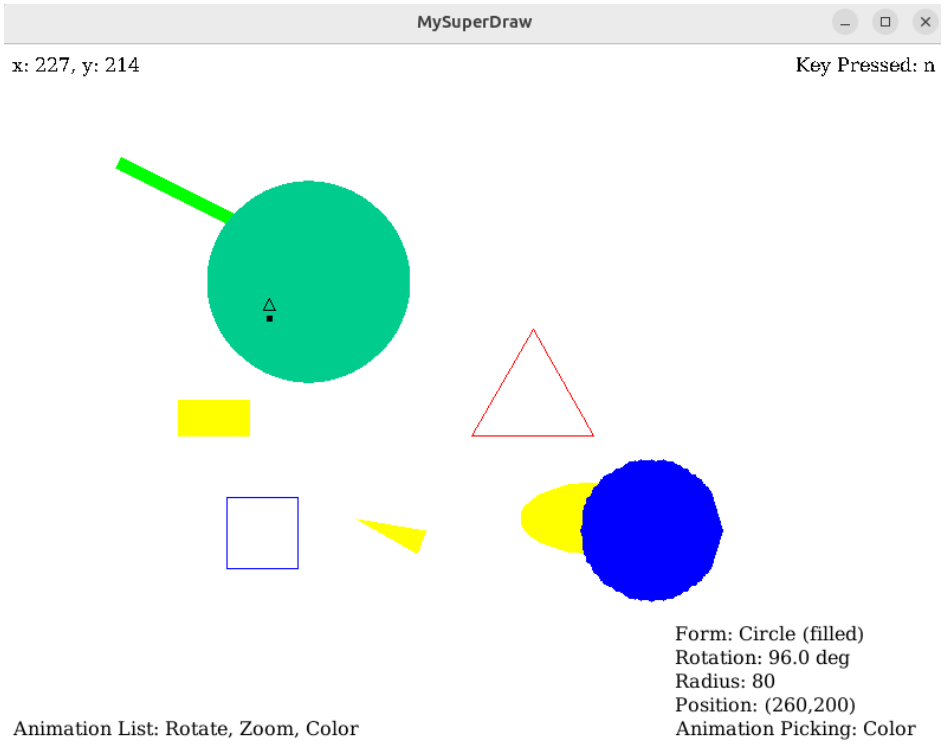

Figure 9: Animations appliquées sur le cercle : rotation, couleur et zoom.

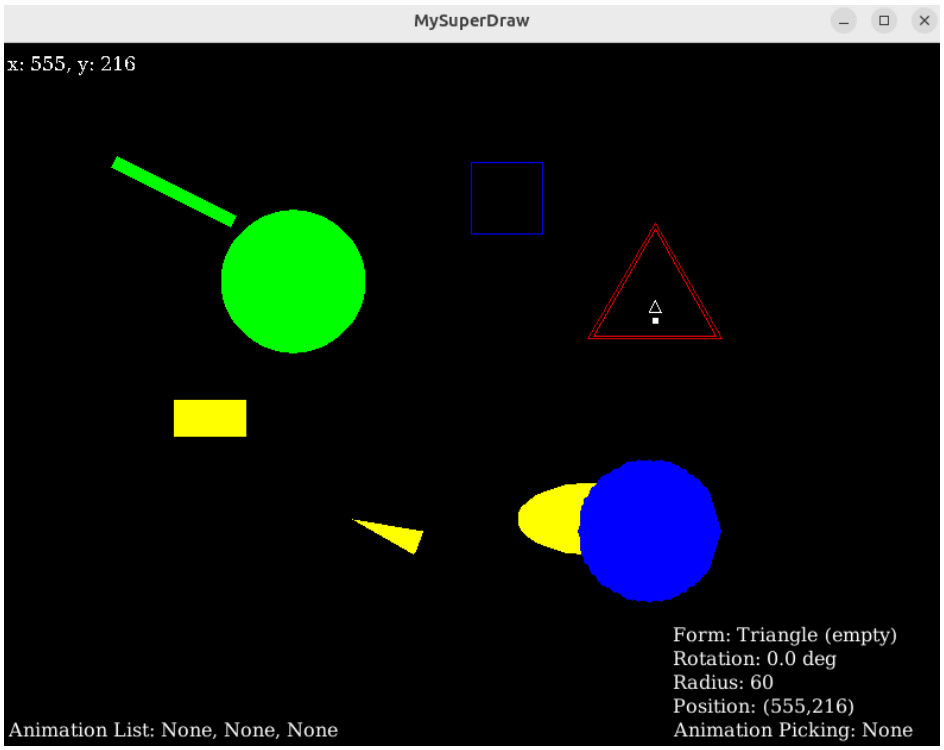

Figure 10: Changement de couleur de la fenêtre.

7 Structure générale de l'IDE

Cet article explique le fonctionnement détaillé de l'IDE **DrawStudioCode**, conçu pour prendre en charge le langage **Draw++**. Il couvre les principales fonctionnalités, la structure du code, et les interactions utilisateur. L'IDE est construit avec **PyQt5** pour l'interface graphique et prend en charge les fonctionnalités suivantes :

- Gestion d'onglets et de plusieurs fenêtre simultanées.
- Éditeur de code avec coloration syntaxique, et lexer/parser intégré
- Terminal intégré pour exécuter et déboguer des scripts.
- Gestion des erreurs avec des suggestions interactives.
- Ouverture du dernier Fichier utilisé
- Changement de nom de page dynamique et détection automatique du besoin de sauvegarde d'un fichier.
- Recherche et remplacement dans l'éditeur.

7.1 Bibliothèques utilisées

Lors de la conception de l'IDE, nous avions initialement envisagé d'utiliser **Tkinter** en raison de sa simplicité. Cependant, nous avons finalement opté pour une bibliothèque beaucoup plus complète et capable de gérer un grand nombre de fonctionnalités de manière native : **PyQt5**. Cette décision nous a permis de bénéficier de fonctionnalités avancées, telles que :

- Gestion des fenêtres et des onglets.
- Intégration native des terminaux et des processus.
- Support pour la coloration syntaxique avancée.
- Personnalisation de l'interface avec des feuilles de style (`QSS`).

—

8 Composants principaux

8.1 CodeEditor

Le composant principal de l'éditeur de code est la classe `CodeEditor`. Cette classe inclut une gestion des numéros de lignes via `LineNumberArea`, une fonctionnalité tirée des forums officiels de PyQt5 et utilisée exclusivement pour afficher les numéros de lignes sur le côté. Ces fonctionnalités ne jouent pas de rôle fonctionnel direct.

8.2 SyntaxHighlighter

La classe `SyntaxHighlighter` est native de **PyQt5**. Elle fonctionne en appliquant une liste de règles, où chaque règle est associée à des mots-clés spécifiques et à un formatage attribué. Les règles de coloration incluent :

- Mots-clés : `var`, `func`, `return`, etc.
- Couleurs : `red`, `green`, `blue`, etc.
- Structures de contrôle : `if`, `else`, `for`, etc.
- Commentaires : Détection des commentaires simples (`#`) et multi-lignes (`/* */`).

—

9 MyDrawppIDE

La classe `MyDrawppIDE` est la classe principale de l'application. Elle représente la fenêtre principale de l'IDE et gère l'ensemble des interactions utilisateur. Voici les principales méthodes et leur rôle :

9.1 `init_ui`

Cette méthode initialise l'interface utilisateur de l'IDE. Elle configure les composants principaux, tels que :

- La barre d'outils avec les actions pour ouvrir, sauvegarder, exécuter, etc.
- Le widget d'onglets pour permettre l'ouverture de plusieurs fichiers simultanément.
- Les raccourcis clavier comme `Ctrl+O` pour ouvrir un fichier et `Ctrl+S` pour sauvegarder.

9.2 open_new_tab

Cette méthode permet de créer un nouvel onglet dans l'IDE. Initialement, nous avions intégré la gestion des nouveaux onglets directement dans la méthode d'initialisation de l'interface utilisateur (`init_ui`). Cependant, cette approche ne permettait de créer qu'une seule instance partagée pour tous les onglets. Cela limitait les fonctionnalités de l'IDE, notamment l'exécution simultanée de plusieurs processus (comme l'exécution ou le débogage de code) dans différents onglets.

Pour surmonter cette limitation, chaque onglet est maintenant conçu pour gérer :

- **Son propre terminal** : Chaque terminal est indépendant et peut afficher les sorties du processus lié à l'onglet correspondant.
- **Son propre processus (`QProcess`)** : Permet l'exécution simultanée de plusieurs scripts sans conflit.

- **Son propre** `SyntaxHighlighter` : Chaque éditeur d'un onglet applique des règles de coloration syntaxique de manière autonome.
- **Son propre** `Editeur` :Chaque onglet à son prorpre éditeur, ce qui assure le lecture indépendante pour la phase de compilation et d'éxécution

Cette conception garantit une modularité accrue et améliore les performances globales de l'IDE lors du travail sur plusieurs projets en parallèle.

9.3 Gestion des fichiers : open_file, save_file, save_as_file

Ces trois méthodes gèrent les fichiers dans l'IDE :

- `open_file` : Ouvre un fichier existant et affiche son contenu dans un nouvel onglet. Si le fichier est déjà ouvert, l'onglet correspondant est sélectionné.
- `save_file` : Sauvegarde le contenu de l'éditeur actif dans le fichier correspondant. Si le fichier n'a pas encore été nommé, la méthode `save_as_file` est appelée pour demander un emplacement et un nom.
- `save_as_file` : Permet de sauvegarder le contenu de l'éditeur actif dans un nouveau fichier. L'utilisateur choisit le nom et l'emplacement via une boîte de dialogue.

9.4 execute_code

Cette méthode est le cœur de l'IDE, gérant l'exécution, la compilation et le débogage du code. L'algorithme fonctionne comme suit :

1. **Écriture dans un fichier temporaire** : Le contenu de l'éditeur actif est d'abord sauvegardé dans un fichier temporaire nommé `to_compile.dpp`. Ce fichier est conçu pour être lu directement par le compilateur.
2. **Analyse de la plateforme** : La méthode détecte le système d'exploitation (Windows ou Unix) afin de construire la commande appropriée pour l'exécution.
3. **Création et gestion du processus** : Un objet `QProcess` est utilisé pour exécuter la commande correspondant au mode spécifié (`run`, `compile` ou `debug`). Ce processus agit comme un terminal fictif.
4. **Affichage des sorties** : Les retours générés par le processus (par exemple, les sorties standard ou les erreurs produites par le compilateur **PLY**) sont récupérés et affichés dans le terminal intégré via la méthode `display_output`.

Cette approche modulaire permet une exécution flexible et assure que chaque mode d'exécution fonctionne de manière isolée, même dans des onglets distincts.

9.5 display_output

La méthode `display_output` est essentielle car elle simule le retour du compilateur directement dans l'IDE. Voici son fonctionnement détaillé :

1. **Connexion au processus** : Dans la méthode `execute_code`, la ligne suivante connecte le signal de lecture des sorties standard du processus à la méthode `display_output` :

```
process.readyReadStandardOutput.connect(
    lambda: self.display_output(process, terminal)
)
```

   Cette connexion permet d'appeler `display_output` chaque fois qu'une sortie est générée par le processus.

2. **Traitement des codes de couleur** : Les sorties du compilateur incluent des codes spéciaux indiquant la couleur à attribuer aux messages :

   - `-#red` : Messages d'erreur, affichés en rouge.
   - `-#green` : Messages de succès, affichés en vert.
   - `-#blue` : Messages d'information, affichés en bleu.

   Ces codes sont analysés dans `display_output`, qui attribue la couleur correspondante à chaque ligne :

3. **Affichage dans le terminal intégré** : La méthode `append_colored_text` est utilisée pour insérer les lignes colorées dans le terminal de l'onglet actif. Cela garantit une expérience utilisateur interactive et visuelle dans l'IDE.

9.6 Soulignement des erreurs syntaxiques

L'IDE met en évidence les erreurs syntaxiques détectées dans l'éditeur grâce à une interaction entre le `QSyntaxHighlighter` de PyQt5 et un analyseur syntaxique personnalisé (basé sur **PLY**). Voici une explication détaillée du processus :

1. **Détection des modifications de texte** : À chaque modification dans l'éditeur, un signal `textChanged` est émis, déclenchant un minuteur différé pour l'analyse syntaxique. Cela permet d'éviter de lancer une analyse après chaque frappe de clavier, réduisant ainsi les ralentissements.

```
self.syntax_check_timer = QTimer(self)
self.syntax_check_timer.setSingleShot(True)
self.syntax_check_timer.timeout.connect(self.check_syntax)
self.textChanged.connect(self.start_syntax_check_timer)
```

   Le minuteur est configuré pour démarrer une analyse syntaxique 100ms après que l'utilisateur a cessé de taper.

2. **Analyse syntaxique différée** : Une fois le délai écoulé, la méthode `check_syntax` est appelée. Voici ses étapes principales :

   - **Analyse lexicale** : Le contenu de l'éditeur est analysé à l'aide d'un analyseur lexical (`lexer`). Si une erreur lexicale est détectée, elle est immédiatement traitée.

```
        lexer = init_lexer()
        lexer.input(text)
        try:
            list(lexer)  # Tokenize the input
        except SyntaxError as e:
            self._handle_error(str(e), error_lines,
               error_messages)
            return
```

   - **Analyse syntaxique** : Si l'analyse lexicale réussit, un analyseur syntaxique (`parser`) est utilisé pour construire un arbre syntaxique abstrait (AST). Toute erreur détectée à ce niveau est également signalée.

```
        parser = init_parser()
        try:
            ast = parser.parse(text, lexer=lexer)
            if ast:
                execute_ast(ast, False, "IDE")
        except Exception as e:
            self._handle_error(str(e), error_lines,
               error_messages)
```

3. **Gestion des erreurs détectées** : Les erreurs détectées sont enregistrées dans une structure contenant :

   - **Les lignes avec erreurs** (`error_lines`) : Un ensemble d'indices des lignes à mettre en évidence.
   - **Les messages d'erreur détaillés** (`error_messages`) : Associés à chaque ligne pour afficher des infobulles d'aide.

   Ces informations sont ensuite transmises au `SyntaxHighlighter` :

```
    if hasattr(self, 'highlighter'):
        self.highlighter.set_error_lines(error_lines)
        self.highlighter.error_messages = error_messages
```

4. **Affichage des erreurs dans l'éditeur** : Le `QSyntaxHighlighter` applique les formats visuels, notamment un soulignement ondulé rouge, pour les lignes contenant des erreurs. Voici un exemple de son fonctionnement :

```
if block_number in self.error_lines:
    error_format = QTextCharFormat()
    error_format.setUnderlineStyle(QTextCharFormat.WaveUnderline)
    error_format.setUnderlineColor(QColor(Qt.red))
    self.setFormat(0, len(text), error_format)
```

   Une infobulle affichant le message d'erreur et une suggestion contextuelle est également associée à chaque ligne via des données utilisateur (`QTextBlockUserData`).

5. **Suggestions automatiques** : En cas d'erreurs comme un identifiant inconnu ou une mauvaise utilisation d'un type, la méthode `_handle_error` génère des suggestions automatiques basées sur des mots-clés connus :

```
if "Unknown identifier" in error_msg:
    unknown_id = error_msg.split("'")[1]
    suggestion = f"Did you mean '{suggested_keyword}'?"
```

   Ces suggestions sont incluses dans les infobulles associées aux lignes contenant des erreurs.

Ce mécanisme garantit une détection rapide et précise des erreurs tout en fournissant une expérience utilisateur intuitive grâce aux infobulles et aux mises en évidence visuelles.

—

10 Conclusion

Le projet Draw++ illustre la puissance des langages spécifiques dans la simplification de tâches complexes comme le dessin et l'animation graphique. Grâce à une syntaxe intuitive et un ensemble d'instructions bien définies, ce langage offre une expérience utilisateur simple et fluide, tout en permettant des possibilités créatives variées.

Au cœur du projet, le traducteur repose sur une architecture robuste intégrant un analyseur lexical et un analyseur syntaxique, soutenus par la bibliothèque **PLY**. Cette approche modulaire a permis de construire un système capable d'interpréter efficacement le code écrit en Draw++, tout en détectant et signalant les erreurs de manière claire et précise.

Les principales réalisations incluent :

- La définition d'une grammaire complète pour Draw++, prenant en charge des fonctionnalités de base (dessin, déplacement, rotation) et avancées (conditions, boucles, animations).
- La mise en place de mécanismes robustes pour l'analyse des erreurs, avec des suggestions pour guider les utilisateurs dans la correction de leur code.
- La création d'un outil flexible et extensible, ouvrant la voie à des améliorations futures.

Perspectives d'amélioration : Bien que le projet ait atteint ses objectifs principaux, plusieurs pistes d'amélioration peuvent être envisagées :

- **Enrichissement des fonctionnalités :** Ajouter de nouvelles formes, des effets graphiques avancés (ombrage, dégradés) ou des fonctionnalités interactives.
- **Optimisation des performances :** Améliorer l'efficacité du traducteur pour gérer des programmes plus complexes ou volumineux.
- **Interface utilisateur :** Intégrer une interface graphique permettant d'écrire, d'exécuter et de visualiser directement les programmes Draw++.
- **Support multi-plateforme :** Adapter Draw++ pour fonctionner sur différentes plateformes, comme le web ou des applications mobiles.

En conclusion, Draw++ constitue une base solide pour l'exploration et la création graphique, tout en servant de projet pédagogique démontrant les principes fondamentaux de la conception de langages et de l'analyse syntaxique. Il ouvre également des perspectives passionnantes pour une utilisation élargie et des développements futurs.