Rapport Draw

Professeur : Zaouche Djaouida

Auteur : Ghoudi Souleim, Sail Rayan, Douici Younes, Sekkat Mehdi,
Nejdar Hadjer
Date : 09/01/2025
Classe : ING-1 GIA

Contents

1 Grammaire du mot-clé `SI` et SINON

Description du mot-clé `SI` et SINON :
Le mot-clé `SI` permet de définir une structure conditionnelle dans le langage Draw. Il permet de spécifier un bloc de code qui sera exécuté uniquement si une condition spécifiée est vraie. Cette condition est généralement une expression qui compare des valeurs (par exemple, une variable à un nombre ou une autre variable) à l'aide d'opérateurs comme `==>`, `<`, `>`, etc.

Si la condition est remplie (c'est-à-dire qu'elle évalue à `vrai`), le bloc de code qui suit le mot-clé `SI` est exécuté. Si la condition est fausse, un autre bloc de code peut être exécuté sous la condition `SINON`. Le mot-clé `SINON` permet donc de spécifier un comportement alternatif lorsque la condition initiale n'est pas remplie. Ce comportement peut inclure un autre bloc de code qui sera exécuté uniquement si la condition du `SI` est fausse.

Structure générale :

`SI` < `expression` > { < `statements` > } [`SINON` { < `statements` > }]

<if-statement> ::= `SI` < `expression` > { `statements` } [`SINON` { `statements` }]

<expression> ::= `variable` | `variable operator value`

<operator> ::= `==>` | `-->` | `<` | `>` | `<=` | `>=` | `+` | `-`

<value> ::= `number` | `string` | `boolean`

Exemple d'utilisation du `SI` dans le code `Draw++` :

```
si x ==>10 {
afficher("X est égal à 10")
}
sinon {
afficher("X n'est pas égal à 10")
}
```

Explication de l'exemple :

1. **Condition :** `x ==>10` Cette condition vérifie si la variable `x` est égale à 10.
2. **Bloc exécuté si la condition est vraie :** Si la condition `x ==>10` est vraie (c'est-à-dire que `x` est égal à 10), le programme exécutera le bloc suivant :

   ```
   afficher("X est égal à 10")
   ```

3. **Bloc exécuté si la condition est fausse (SINON) :** Si la condition est fausse (c'est-à-dire que `x` n'est pas égal à 10), le programme exécutera le bloc suivant :

   ```
   afficher("X n'est pas égal à 10")
   ```

Comment cela fonctionne dans un programme `Draw++` :

Ce genre de structure conditionnelle est utile pour effectuées des actions en fonction de certaines conditions. Par exemple, on pourrait utiliser cette condition pour décider si une forme doit être dessinée à une position donnée ou si le curseur doit être déplacé à un endroit spécifique en fonction de la valeur d'une variable. Elle permet ainsi de contrôler dynamiquement les actions dans un programme en fonction des données d'entrée ou des calculs effectués.

2 Grammaire du mot-clé `POUR`

Description du mot-clé `POUR` :

Le mot-clé `POUR` permet de définir une structure itérative dans le langage `Draw`. Cette structure est utilisée pour répéter un bloc de code un nombre défini de fois. L'instruction `POUR` spécifie une variable de contrôle, une plage de valeurs, et un bloc de code à exécuter pour chaque valeur dans cette plage. La plage de valeurs est définie par deux bornes : une borne de début (après le mot-clé `DE`) et une borne de fin (après le mot-clé `A`).

Structure générale :

```
POUR < variable > DE < expression-début > A < expression-fin
> { < statements > }
```

Grammaire formelle :

<for-statement> ::= POUR < variable > DE < expression-début > A < expression-fin > { statements }

<expression-début> ::= value | variable

<expression-fin> ::= value | variable

<statements> ::= < instructions >

<variable> ::= nom-d'une-variable-valide

<value> ::= number | boolean

Exemple d'utilisation du mot-clé `POUR` :

```
pour i de 1 à 5 {
afficher("Bonjour")
}
```

Explication de l'exemple :

1. **Variable de contrôle :** La variable `i` est utilisée pour contrôler l'itération.
2. **Plage de valeurs :** La boucle commence à `1` (borne de début) et se termine à `5` (borne de fin).
3. **Bloc exécuté :** Le bloc de code à l'intérieur des accolades {...} est exécuté pour chaque valeur de `i` dans la plage. Ici, il affiche la valeur actuelle de `i`.

Utilité dans un programme `Draw++` : Cette structure est essentielle pour automatiser des actions répétées, comme dessiner une série de formes, déplacer des objets de manière incrémentale, ou appliquer des transformations multiples à des éléments visuels. Par exemple, une boucle `POUR` pourrait être utilisée pour tracer un cercle composé de 10 points espacés régulièrement.

3 Grammaire du mot-clé `TANTQUE`

Description du mot-clé `TANTQUE` :
Le mot-clé `TANTQUE` permet de définir une structure de boucle conditionnelle dans le langage `Draw`. La boucle continue à s'exécuter tant que la condition spécifiée est vraie. La condition est une expression qui doit être évaluée avant chaque itération de la boucle. Tant que la condition évalue à `vrai`, le bloc de code associé à la boucle sera exécuté.

Structure générale :

```
TANTQUE < condition > { < statements > }
```

Grammaire formelle :

<while-statement> ::= `TANTQUE < condition > { statements }`

<condition> ::= `expression-condition | variable | boolean`

<statements> ::= `< instructions >`

<expression-condition> ::= `variable | variable operator value`

<operator> ::= `==> | --> | < | > | <= | >= | + | -`

<value> ::= `number | boolean`

Exemple d'utilisation du mot-clé `TANTQUE` :

```
tantque x < 10 {
x = x + 1
}
```

Explication de l'exemple :

1. **Condition :** La boucle continue tant que `x < 10` est vrai.
2. **Bloc exécuté :** À chaque itération, la valeur de `x` est augmentée de 1. La boucle s'exécutera jusqu'à ce que `x` atteigne 10.

Utilité dans un programme `Draw++` : La structure `TANTQUE` est idéale pour exécuter une série d'actions répétitives tant qu'une condition est remplie. Cela peut être utile dans des situations où il est nécessaire de répéter une opération jusqu'à ce qu'une certaine condition d'arrêt soit atteinte, comme par exemple déplacer un objet jusqu'à ce qu'il atteigne une position donnée.

4 Grammaire des opérateurs ==> et ->

Description des opérateurs :
Dans le langage `Draw++`, les opérateurs `==>` et `->` ont des rôles distincts : - `==>` est utilisé pour effectuer une comparaison d'équivalence, équivalente à `==` dans d'autres langages. - `->` est utilisé pour l'assignation de valeurs, équivalente à `=` dans d'autres langages.

Ces opérateurs sont intégrés dans les expressions et permettent une manipulation fluide des données dans le langage.

—

Grammaire formelle :

<expression> ::= <primary> [<operator> <primary>]

<operator> ::= ==> | -> | < | > | <= | >= | + | -

<primary> ::= variable | value

<assignment> ::= variable -> <expression>

<binary-expression> ::= <primary> ==> <primary>

—

Exemple d'utilisation :

Comparaison avec ==> :

```
si x ==> 10 {
afficher("x est égal à 10")
}
```

Assignation avec -> :

```
x -> 10
```

—

Explication des exemples :

1. **Comparaison :** L'expression `x ==> 10` vérifie si la variable `x` est égale à 10. Si cette condition est vraie, le programme exécute le bloc suivant.
2. **Assignation :** L'expression `x -> 10` attribue la valeur `10` à la variable `x`.

—

Utilité dans un programme `Draw++` :

Ces opérateurs permettent de gérer les données et de contrôler le flux d'exécution. - Les comparaisons (`==>`) sont utiles pour les conditions dans les structures comme `SI`, `TANTQUE`, ou `POUR`. - L'assignation (`->`) permet de définir et modifier les valeurs des variables tout au long du programme.

5 Grammaire de la commande `afficher`

Description de la commande `afficher` :
La commande `afficher` permet d'afficher des informations à l'écran. Elle commence par le mot-clé `texte`, suivi de parenthèses contenant une ou plusieurs expressions. Les expressions peuvent être des chaînes de texte, des variables ou des résultats d'opérations. Les expressions sont séparées par des virgules, mais ne peuvent pas être vides.

—

Grammaire formelle :

<print-statement> ::= texte (<expression-list>)

<expression-list> ::= <expression> [, <expression>]*

<expression> ::= chaine | variable | valeur | <binary-expression>

<chaine> ::= "texte"

<binary-expression> ::= <expression> <operator> <expression>

<operator> ::= ==> | + | - | < | >

—

Exemple d'utilisation :

```
texte("La valeur de x est ", x, " et égale à ", 10)
```

—

Explication de l'exemple :

1. **Commande :** La commande `texte` est utilisée pour afficher des informations.
2. **Expressions à imprimer :** - `"La valeur de x est "` : Chaîne de texte. - `x` : Une variable. - `" et égale à "` : Chaîne de texte. - `10` : Valeur numérique.
3. **Résultat attendu :** Si `x = 10`, le programme affichera : `La valeur de x est 10 et égale à 10`

—

Utilité dans un programme `Draw++` :

La commande `texte` est essentielle pour afficher des informations à l'utilisateur pendant l'exécution du programme. Elle permet de combiner des chaînes de texte et des variables pour créer des messages dynamiques et interactifs.

6 Grammaire de la commande `drawLine`

Description de la commande `drawLine` :
La commande `drawLine` permet de dessiner une ligne entre deux points spécifiés par leurs coordonnées (x_1, y_1) et (x_2, y_2). Les coordonnées sont des expressions qui peuvent être des variables, des nombres, ou des résultats d'opérations. Les arguments doivent être séparés par des virgules.

—

Grammaire formelle :

<drawLine-statement> ::= drawLine (<expression> , <expression> , <expression> , <expression>)

<expression> ::= variable | number | <binary-expression>

<binary-expression> ::= <expression> <operator> <expression>

<operator> ::= + | - | * | /

—

Exemple d'utilisation :

```
drawLine(10, 20, 30, 40)
```

—

Explication de l'exemple :

1. **Commande :** La commande `drawLine` est utilisée pour dessiner une ligne entre deux points.

2. **Arguments :** - $x_1 = 10$: Coordonnée x du premier point. - $y_1 = 20$: Coordonnée y du premier point. - $x_2 = 30$: Coordonnée x du second point. - $y_2 = 40$: Coordonnée y du second point.

3. **Résultat attendu :** Le programme dessinera une ligne entre les points $(10, 20)$ et $(30, 40)$.

—

Utilité dans un programme `Draw++` :

La commande `drawLine` est utilisée pour dessiner des formes et des graphiques en reliant deux points par une ligne droite. Elle peut être combinée avec d'autres commandes pour créer des graphiques complexes.

7 Grammaire de la commande `drawSquare`

Description de la commande `drawSquare` :
La commande `drawSquare` permet de dessiner un carré dont la position de départ (coin supérieur gauche) et la taille sont spécifiées. Les arguments sont des expressions pouvant être des variables, des nombres, ou des résultats d'opérations. Les arguments doivent être séparés par des virgules.

—

Grammaire formelle :

<drawSquare-statement> ::= `drawSquare ( <expression> , <expression> , <expression> )`

<expression> ::= `variable` | `number` | `<binary-expression>`

<binary-expression> ::= `<expression> <operator> <expression>`

<operator> ::= `+` | `-` | `*` | `/`

—

Exemple d'utilisation :

```
drawSquare(10, 20, 30)
```

—

Explication de l'exemple :

1. **Commande :** La commande `drawSquare` est utilisée pour dessiner un carré avec un coin supérieur gauche à la position spécifiée et une taille donnée.

2. **Arguments :** - $x = 10$: Coordonnée x du coin supérieur gauche. - $y = 20$: Coordonnée y du coin supérieur gauche. - $size = 30$: Taille du carré (côté du carré).

3. **Résultat attendu :** Le programme dessinera un carré avec son coin supérieur gauche à $(10, 20)$ et un côté de taille 30.

—

Utilité dans un programme `Draw++` :

La commande `drawSquare` est utilisée pour dessiner un carré, ce qui peut être utile pour créer des éléments graphiques de base dans un environnement de dessin. Elle peut être utilisée pour dessiner des formes géométriques simples et pour créer des interfaces visuelles.

8 Grammaire de la commande `drawCircle`

Description de la commande `drawCircle` :
La commande `drawCircle` permet de dessiner un cercle dont le centre et le rayon sont spécifiés. Les arguments sont des expressions pouvant être des variables, des nombres, ou des résultats d'opérations. Les arguments doivent être séparés par des virgules.

—

Grammaire formelle :

<drawCircle-statement> ::= `drawCircle ( <expression> , <expression> , <expression> )`

<expression> ::= `variable` | `number` | **<binary-expression>**

<binary-expression> ::= **<expression>** **<operator>** **<expression>**

<operator> ::= `+` | `-` | `*` | `/`

—

Exemple d'utilisation :

```
drawCircle(10, 20, 30)
```

—

Explication de l'exemple :

1. **Commande :** La commande `drawCircle` est utilisée pour dessiner un cercle avec un centre spécifié par les coordonnées x et y, et un rayon donné.

2. **Arguments :** - $x = 10$: Coordonnée x du centre du cercle. - $y = 20$: Coordonnée y du centre du cercle. - $radius = 30$: Rayon du cercle.

3. **Résultat attendu :** Le programme dessinera un cercle avec son centre en $(10, 20)$ et un rayon de 30.

—

Utilité dans un programme `Draw++` :

La commande `drawCircle` permet de dessiner des formes circulaires. Elle peut être utilisée pour illustrer des concepts géométriques, des interfaces graphiques ou des représentations visuelles dans un programme.

9 Grammaire de la commande `drawArc`

Description de la commande `drawArc` :
La commande `drawArc` permet de dessiner un arc de cercle dont le centre, le rayon, ainsi que les angles de début et de fin sont spécifiés. Les arguments doivent être

séparés par des virgules, et peuvent être des variables, des nombres, ou des résultats d'opérations.

—

Grammaire formelle :

<drawArc-statement> ::= drawArc (<expression> , <expression> , <expression> , <expression> , <expression>)

<expression> ::= variable | number | <binary-expression>

<binary-expression> ::= <expression> <operator> <expression>

<operator> ::= + | - | * | /

—

Exemple d'utilisation :

```
drawArc(10, 20, 30, 0, 90)
```

—

Explication de l'exemple :

1. **Commande :** La commande `drawArc` est utilisée pour dessiner un arc de cercle avec un centre spécifié par les coordonnées x et y, un rayon donné, ainsi que des angles de départ et de fin.

2. **Arguments :** - $x = 10$: Coordonnée x du centre de l'arc. - $y = 20$: Coordonnée y du centre de l'arc. - $radius = 30$: Rayon de l'arc. - $start\_angle = 0$: Angle de départ de l'arc (en degrés ou radians, selon la convention). - $end\_angle = 90$: Angle de fin de l'arc.

3. **Résultat attendu :** Le programme dessinera un arc de cercle avec son centre en $(10, 20)$, un rayon de 30, et couvrant les angles de 0° à 90°.

—

Utilité dans un programme `Draw++` :

La commande `drawArc` permet de dessiner des arcs de cercle, ce qui peut être utile pour des graphiques, des représentations de courbes, des diagrammes ou des interfaces graphiques.

10 Grammaire de la commande `drawCursor`

Description de la commande `drawCursor` :
La commande `drawCursor` permet de déplacer un curseur graphique à une position donnée sur l'écran. Les arguments spécifient la position x et y du curseur.

—

Grammaire formelle :

<drawCursor-statement> ::= drawCursor (<expression> , <expression>)

<expression> ::= variable | number | <binary-expression>

<binary-expression> ::= <expression> <operator> <expression>

<operator> ::= + | - | * | /

—

Exemple d'utilisation :

```
drawCursor(50, 100)
```

—

Explication de l'exemple :

1. **Commande :** La commande `drawCursor` est utilisée pour déplacer un curseur graphique à une position spécifiée par les coordonnées x et y.

2. **Arguments :** - $x = 50$: Coordonnée x pour la position du curseur. - $y = 100$: Coordonnée y pour la position du curseur.

3. **Résultat attendu :** Le programme déplacera le curseur à la position $(50, 100)$ sur l'écran.

—

Utilité dans un programme `Draw++` :

La commande `drawCursor` est utilisée pour déplacer un curseur dans une interface graphique ou dans une animation, permettant ainsi de suivre des déplacements ou de pointer des éléments sur l'écran.

11 Grammaire de la commande `moveCursor`

Description de la commande `moveCursor` :
La commande `moveCursor` permet de déplacer un curseur sur l'écran en fonction des déplacements relatifs dx et dy par rapport à sa position actuelle. Les arguments spécifient les déplacements relatifs en x et en y.

—

Grammaire formelle :

<moveCursor-statement> ::= moveCursor (<expression> , <expression>)

<expression> ::= variable | number | <binary-expression>

<binary-expression> ::= <expression> <operator> <expression>

<operator> ::= + | - | * | /

—

Exemple d'utilisation :

```
moveCursor(10, -5)
```

—

Explication de l'exemple :

1. **Commande :** La commande `moveCursor` est utilisée pour déplacer un curseur sur l'écran selon les déplacements relatifs dx et dy.

2. **Arguments :** - $dx = 10$: Déplacement relatif en x. - $dy = -5$: Déplacement relatif en y.

3. **Résultat attendu :** Le programme déplacera le curseur de 10 unités vers la droite et de 5 unités vers le bas.

—

Utilité dans un programme `Draw++` :

La commande `moveCursor` est utilisée pour déplacer un curseur de façon dynamique sur l'écran, permettant ainsi de suivre des animations ou de pointer des éléments relatifs à la position actuelle du curseur.

12 Grammaire de la commande `rotateCursor`

Description de la commande `rotateCursor` :
La commande `rotateCursor` permet de faire pivoter le curseur autour de sa position actuelle. L'argument spécifie l'angle de rotation en degrés.

—

Grammaire formelle :

<rotateCursor-statement> ::= `rotateCursor ( <expression> )`

<expression> ::= `variable` | `number` | `<binary-expression>`

<binary-expression> ::= `<expression> <operator> <expression>`

<operator> ::= `+` | `-` | `*` | `/`

—

Exemple d'utilisation :

```
rotateCursor(45)
```

—

Explication de l'exemple :

1. **Commande :** La commande `rotateCursor` est utilisée pour faire pivoter le curseur de l'écran d'un certain angle.

2. **Argument :** - angle = 45 : L'angle de rotation en degrés.

3. **Résultat attendu :** Le programme effectuera une rotation du curseur de 45 degrés dans le sens horaire ou antihoraire, selon la convention définie par l'implémentation.

—

Utilité dans un programme `Draw++` :

La commande `rotateCursor` permet de faire pivoter un curseur de manière précise, ce qui peut être utile pour dessiner des formes ou manipuler des objets dans des programmes interactifs ou des applications graphiques.

13 Analyse de la fonction `tokenize`

La fonction `tokenize` est chargée de convertir une chaîne de caractères contenant du code source en une liste de jetons (*tokens*). Ces jetons sont des unités lexicales qui représentent les éléments syntaxiques du code, tels que des mots-clés, des opérateurs ou des identifiants. Voici une description détaillée de son fonctionnement :

- **Initialisation :** La fonction commence par récupérer le code source à analyser et initialise une liste vide `tokens` pour stocker les jetons détectés.
- **Boucle principale :** Tant qu'il reste du code à analyser, la fonction parcourt la liste des modèles de jetons (`self.token_patterns`). Pour chaque modèle, elle essaie de trouver une correspondance au début du code en utilisant des expressions régulières.
- **Correspondance trouvée :**
  - Si un modèle correspond, la valeur correspondante est extraite avec `match.group(0)`.
  - Si le type de jeton n'est pas `ESPACE`, le jeton est ajouté à la liste `tokens` sous la forme d'un tuple (`token_type, value`).
  - Le code est raccourci en supprimant la partie correspondant au jeton détecté (`code[len(value):]`).
- **Aucune correspondance :** Si aucun modèle ne correspond au début du code, une exception `SyntaxError` est levée, signalant la présence d'un caractère inattendu.
- **Fin de l'analyse :** Une fois tout le code analysé, un jeton de fin de fichier (`EOF`) est ajouté à la liste des jetons pour marquer la fin de l'analyse.
- **Retour des jetons :** La fonction retourne la liste complète des jetons détectés.

Cette approche permet d'analyser le code source de manière incrémentale, en découpant le texte en éléments compréhensibles pour les étapes ultérieures du traitement syntaxique ou sémantique.

La fonction `consume`

La fonction `consume` est une méthode clé dans le processus d'analyse syntaxique. Elle est utilisée pour consommer (*consume*) un jeton de la liste et vérifier, si nécessaire, qu'il correspond au type attendu. Cette vérification garantit que les jetons sont analysés dans un ordre conforme à la grammaire du langage.

Fonctionnement

- **Lecture du jeton courant :** La fonction commence par récupérer le jeton situé à la position actuelle, désigné par l'attribut `self.pos`.
- **Vérification du type attendu :** Si un type attendu est spécifié en argument (`expected_type`), la fonction compare ce type avec celui du jeton courant. Si les types ne correspondent pas, une exception `SyntaxError` est levée avec un message explicatif contenant le type attendu et le type reçu.
- **Avancement :** En cas de correspondance (ou si aucun type attendu n'est spécifié), la position actuelle (`self.pos`) est incrémentée pour avancer au jeton suivant.
- **Retour :** Le jeton consommé est renvoyé, permettant aux autres parties du programme de l'utiliser pour des analyses ou traitements ultérieurs.

Exemple de fonctionnement

Supposons que la liste de jetons contienne `[("NOMBRE", "5"), ("OPERATEUR", "+")]`. Si la fonction est appelée avec `expected_type="NOMBRE"`, elle :

1. Vérifie que le jeton courant a bien pour type `"NOMBRE"`.
2. Incrémente `self.pos` pour passer au jeton suivant.
3. Retourne le jeton `("NOMBRE", "5")`.

Importance dans le programme

La fonction `consume` est fondamentale pour le fonctionnement du parseur. Elle garantit que :

- Les jetons sont analysés dans un ordre logique et conforme à la grammaire.
- Les erreurs de syntaxe sont détectées rapidement lorsque des jetons inattendus sont rencontrés.
- L'avancement dans la liste de jetons est effectué de manière contrôlée, permettant un traitement progressif et structuré du code source.

Cette méthode constitue donc le cœur du mécanisme d'analyse syntaxique et est utilisée comme brique de base pour des fonctions plus complexes dans le parseur.

14 Parseur Syntaxique

Dans le cadre du projet `Draw++`, le parseur syntaxique joue un rôle fondamental dans l'interprétation du code utilisateur. Il intervient après l'analyse lexicale et vérifie que les séquences de jetons générées respectent les règles syntaxiques du langage. Ce parseur est spécifiquement conçu pour traduire les instructions utilisateur dans une structure hiérarchique appelée *Abstract Syntax Tree* (AST), facilitant ainsi l'exécution des commandes.

15 Fonctionnement du parseur syntaxique

Le parseur syntaxique analyse les instructions ligne par ligne pour vérifier leur validité structurelle et les organiser dans un format logique. Il utilise des fonctions dédiées pour reconnaître différents types d'instructions comme les conditions (`SI...SINON`), les boucles (`POUR`, `TANTQUE`) et les commandes spécifiques au dessin (`DRAW_LINE`, `DRAW_SQUARE`, etc.).

Exemple : la fonction `parse_statement`

La fonction `parse_statement` est une pièce maîtresse du parseur syntaxique. Elle identifie le type d'instruction en examinant le prochain jeton (`peek()`) et redirige l'analyse vers une fonction spécifique :

- `parse_if()` pour les conditions `SI...SINON`.
- `parse_for()` pour les boucles `POUR`.
- `parse_draw_line()` ou `parse_draw_square()` pour les commandes de dessin.
- `parse_assignment()` pour les assignations de variables.
- `parse_expression()` pour des expressions autonomes comme des calculs ou des valeurs littérales.

Cette structure modulaire garantit une analyse précise et organisée des instructions. Si un jeton inattendu est rencontré, une exception est levée, signalant une erreur syntaxique.

La construction de l'AST

Chaque fonction de parsing retourne une structure JSON qui représente un nœud dans l'AST. Par exemple :

- La fonction `parse_if()` produit un nœud `IfStatement` contenant une condition, un bloc *then* et, éventuellement, un bloc *else*.
- La fonction `parse_draw_line()` retourne un nœud contenant les coordonnées de début et de fin de la ligne à dessiner.

Voici un exemple d'AST généré pour une instruction `SI` :

```
{
    "type": "IfStatement",
    "condition": {
        "type": "BinaryExpression",
        "operator": "==",
        "left": {"type": "Variable", "name": "x"},
        "right": {"type": "Literal", "value": 10}
    },
    "then": [
        {"type": "PrintStatement", "values": [{"type": "Variable", "name": "x"}]}
    ],
    "else": null
}
```

16 Les commandes spécifiques au dessin

`Draw++` se distingue par ses fonctionnalités graphiques. Le parseur syntaxique gère des commandes comme `DRAW_LINE`, `DRAW_SQUARE` et `DRAW_CIRCLE`. Par exemple, la fonction `parse_draw_line()` s'assure que la commande est suivie de paramètres corrects entre parenthèses, tels que les coordonnées (`x1`, `y1`, `x2`, `y2`). Elle construit ensuite un nœud de type `DRAW_LINE` dans l'AST.

Exemple d'AST pour une commande de dessin :

```
{
    "type": "DRAW_LINE",
    "params": [
        {"type": "Literal", "value": 0},
        {"type": "Literal", "value": 0},
        {"type": "Literal", "value": 100},
        {"type": "Literal", "value": 100}
    ]
}
```

17 Limites et objectifs du parseur syntaxique

Le parseur syntaxique de `Draw++` se concentre uniquement sur la validation de la structure des instructions. Il ne prend pas en charge l'analyse sémantique, ce qui signifie qu'il ne vérifie pas la cohérence logique des opérations (par exemple, dessiner une ligne avec des valeurs incohérentes comme `x1 > x2`). Ces vérifications sont effectuées à des étapes ultérieures, comme l'exécution ou l'analyse sémantique.

18 Analyse sémantique dans `Draw++`

Introduction à l'analyse sémantique

L'analyse sémantique est une étape cruciale après le parsing syntaxique. Si le parseur syntaxique se concentre sur la structure et la validité grammaticale du code,

l'analyse sémantique vérifie la cohérence logique et les règles spécifiques au domaine d'application. Dans `Draw++`, cette étape garantit que les instructions graphiques ou logiques sont non seulement valides syntaxiquement, mais également cohérentes d'un point de vue contextuel.

Objectifs de l'analyse sémantique

L'analyse sémantique dans `Draw++` remplit plusieurs fonctions essentielles :

- Vérifier les types des opérandes : s'assurer que les opérations sont effectuées sur des types compatibles (par exemple, additionner deux nombres).
- Valider les plages de valeurs : s'assurer que les coordonnées ou dimensions graphiques sont valides (par exemple, une position négative pourrait ne pas être autorisée dans certaines situations).
- Détecter les variables non déclarées ou non initialisées.
- Éviter les boucles infinies ou les conditions impossibles dans certaines constructions logiques.

Mécanisme d'analyse sémantique

Dans `Draw++`, l'analyse sémantique intervient sur l'AST produit par le parseur syntaxique. Elle parcourt cet arbre pour vérifier chaque nœud en appliquant des règles sémantiques spécifiques. Voici quelques exemples concrets de validations sémantiques réalisées dans `Draw++`.

Validation des variables

Lorsqu'un nœud de type `Assignment` ou `Variable` est rencontré dans l'AST, l'analyseur sémantique vérifie :

- Si la variable a été déclarée avant son utilisation.
- Si la valeur assignée à la variable est compatible avec son type.

Exemple de détection d'une erreur : Si l'utilisateur tente d'exécuter le code suivant :

```
x = 10;
y = x + "text";
```

L'analyse sémantique signalera une incompatibilité de type entre un entier (`x`) et une chaîne de caractères (`"text"`).

Vérification des commandes graphiques

Les commandes spécifiques à `Draw++`, comme `DRAW_LINE` ou `DRAW_SQUARE`, nécessitent des validations sémantiques précises. Par exemple :

- `DRAW_LINE` : Vérifie que les coordonnées fournies sont des nombres valides.

- `DRAW_SQUARE` : Vérifie que la taille du carré est positive.
- `ROTATE_CURSOR` : S'assure que l'angle de rotation est une valeur numérique.

Exemple de validation sémantique pour une commande `DRAW_LINE` :

```
{
    "type": "DRAW_LINE",
    "params": [
        {"type": "Literal", "value": 0},
        {"type": "Literal", "value": -10},  # Erreur détectée : valeur négative
        {"type": "Literal", "value": 100},
        {"type": "Literal", "value": 100}
    ]
}
```

Dans ce cas, l'analyse sémantique signalerait que les coordonnées `y1` ne peuvent pas être négatives.

Vérification des boucles

Pour les constructions logiques comme `POUR` ou `TANTQUE`, l'analyse sémantique vérifie :

- Que la condition de terminaison est atteignable.
- Que les variables utilisées dans la boucle sont correctement déclarées et modifiées dans le corps de la boucle.

Par exemple, pour une boucle `POUR` mal définie :

```
POUR i DE 10 A 5 { ... }
```

L'analyse sémantique signalera que la plage est invalide, car le début (`10`) est supérieur à la fin (`5`).

Retour d'erreurs sémantiques

Lorsque des erreurs sémantiques sont détectées, `Draw++` retourne des messages clairs à l'utilisateur pour faciliter le débogage. Ces messages incluent des informations sur l'emplacement de l'erreur dans le code source et la nature du problème.

Exemple d'erreur :

```
Erreur sémantique : variable "x" utilisée avant d'être initialisée. Ligne 3.
```

Limites et évolutions possibles

Le parseur sémantique actuel de `Draw++` traite principalement des erreurs simples de type ou de plage. Cependant, certaines améliorations pourraient être envisagées :

- Ajouter un système de types plus avancé, comme la prise en charge des tableaux ou des objets.

- Intégrer une vérification plus poussée des dépendances logiques dans les boucles et les conditions imbriquées.
- Développer des mécanismes de détection automatique des boucles infinies dans des cas complexes.

19 Parseur Lexical

Le parseur lexical présenté ici est une partie essentielle d'un compilateur ou d'un interpréteur. Il prend en entrée une liste de jetons (tokens) générés par un analyseur lexical (lexer) et produit une structure arborescente représentant le programme source. Ce parseur est conçu pour analyser des instructions simples dans un langage de programmation, telles que des expressions, des affectations et des boucles.

20 Fonctions Principales

Le parseur est structuré autour de plusieurs fonctions qui permettent de traiter les jetons un par un et de les organiser en structures plus complexes.

La fonction `consume`

La fonction `consume` permet de consommer le jeton actuel et de vérifier si ce jeton correspond à un type attendu. Si le type de jeton ne correspond pas à celui attendu, une erreur de syntaxe est levée. Sinon, le parseur passe au jeton suivant. Cette fonction est utilisée pour avancer dans le processus de parsing en assurant que les jetons sont correctement traités.

```
def consume(self, expected_type=None):
    current_token = self.tokens[self.pos]
    if expected_type and current_token[0] != expected_type:
        raise SyntaxError(f"Expected {expected_type}, got {current_token[0]}")
    self.pos += 1
    return current_token
```

La fonction `peek`

La fonction `peek` permet de jeter un coup d'œil au jeton actuel sans le consommer. Cela permet au parseur de vérifier le jeton sans avancer la position dans la liste des jetons. Cette fonction est utilisée dans des cas où il faut analyser un jeton sans le retirer de la séquence.

```
def peek(self):
    return self.tokens[self.pos]
```

La fonction `parse`

La fonction `parse` est le point d'entrée principal du parseur. Elle appelle la fonction `parse_statements` pour commencer l'analyse des instructions du programme source.

```
def parse(self):
    return self.parse_statements()
```

La fonction `parse_statements`

La fonction `parse_statements` analyse les instructions du programme en boucle, en s'assurant que chaque instruction est traitée jusqu'à ce que l'on atteigne un jeton de fin de programme (`EOF`) ou une accolade fermante (`ACCOLADE_FERM`).

```
def parse_statements(self):
    statements = []
    while self.peek()[0] not in ("EOF", "ACCOLADE_FERM"):
        statements.append(self.parse_statement())
    return {"type": "Program", "body": statements}
```

La fonction `parse_statement`

La fonction `parse_statement` identifie le type d'instruction à analyser en fonction du jeton courant. Elle supporte plusieurs types d'instructions, y compris les structures de contrôle comme `SI` (si), `POUR` (pour), et `TANTQUE` (tant que), ainsi que des commandes graphiques telles que `DRAW_LINE` (dessiner une ligne) et `MOVE_CURSOR` (déplacer le curseur).

```
def parse_statement(self):
    token = self.peek()
    if token[0] == "SI":
        return self.parse_if()
    elif token[0] == "DRAW_LINE":
        return self.parse_draw_line()
    ...
```

Explication de la fonction `translate` de la classe `CTranslator`

La fonction `translate` est une méthode centrale de la classe `CTranslator`. Elle reçoit en paramètre un nœud d'AST (Abstract Syntax Tree) et génère le code C correspondant. Voici une explication des différentes branches de la fonction, selon le type du nœud analysé.

1. Type `Program`

- La fonction vérifie si le type du nœud est `Program`.
- Elle parcourt tous les sous-nœuds de `body` et les traduit récursivement.
- Le code généré est une concaténation des traductions de chaque sous-nœud, séparée par des sauts de ligne.

```
if ast["type"] == "Program":
    return "\n".join(self.translate(statement) for
       statement in ast["body"])
```

2. Type `IfStatement`

- Cette branche traduit les instructions conditionnelles.
- La condition et le bloc `then` sont traduits récursivement.
- Si un bloc `else` existe, il est également traduit et ajouté au code.

```
elif ast["type"] == "IfStatement":
    code = f"if ({self.translate(ast['condition'])}){{\n{
       self.translate(ast['then']).replace('\n', '\n')}\n
       }}"
    if ast.get("else"):
        code += f"\nelse {{\n    {self.translate(ast['
           else']).replace('\n', '\n    ')}\n}}"
    return code
```

3. Type `Assignment`

- Cette branche traite les affectations de variables.
- Si la variable a déjà été définie, une simple affectation est effectuée.
- Sinon, le type de la valeur est analysé (`Literal`, `Variable`, `BinaryExpression`) pour choisir le type approprié (`int`, `float`, `auto`).

```
elif ast["type"] == "Assignment":
    if ast['variable'] in variable_already_defined:
        return f"{ast['variable']} = {self.translate(ast
           ['value'])};"
    else:
        # Gestion des nouveaux types
        ...
```

4. Type `PrintStatement`

- Génère des instructions `printf` pour afficher une chaîne ou une chaîne suivie de la valeur d'une variable.
- Si plusieurs valeurs sont présentes, la seconde est traduite comme une variable à afficher.

```
elif ast["type"] == "PrintStatement":
    values = self.translate(ast["values"][0]).strip('"')
    if len(ast["values"]) > 1:
        variable = self.translate(ast["values"][1])
        return f'printf("{values}\\n", {variable});'
    else:
        return f'printf("{values}\\n");'
```

5. Type `ForLoop`

- Traduit les boucles `for`.
- Les limites (`start` et `end`) ainsi que le corps de la boucle sont traduits récursivement.
- Le code généré suit la syntaxe classique de C :

```
elif ast["type"] == "ForLoop":
    return f"for (int {variable} = {start}; {variable} <=
        {end}; {variable}++) {{\n    {body}\n}}"
```

6. Type `WhileLoop`

- Gère les boucles `while`.
- La condition et le corps sont traduits récursivement pour générer un code standard en C.

```
elif ast["type"] == "WhileLoop":
    return f"while ({self.translate(ast['condition'])})
       {{\n    {body}\n}}"
```

7. Types Simples : `BinaryExpression`, `Literal`, `Variable`

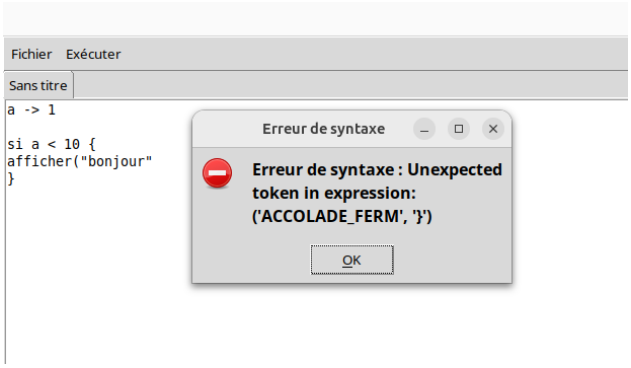
- `BinaryExpression` : Combine les opérandes gauche et droit avec l'opérateur pour créer une expression valide.
- `Literal` : Retourne directement la valeur littérale sous forme de chaîne.
- `Variable` : Retourne simplement le nom de la variable.

```
elif ast["type"] == "BinaryExpression":
    return f"{left} {operator} {right}"

elif ast["type"] == "Literal":
    return str(ast["value"])

elif ast["type"] == "Variable":
    return ast["name"]
```

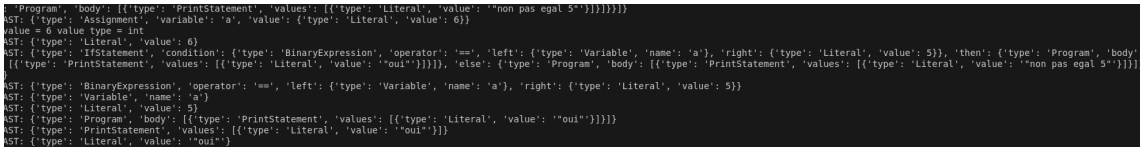
21 Exemple d'erreur



Cette image illustre un exemple de gestion d'erreur dans le langage `Draw++`. L'erreur est causée par une parenthèse fermante manquante dans le code, ce qui déclenche un message d'erreur spécifique.

22 Annexes

Représentation AST



L'image ci-dessus illustre l'AST (Abstract Syntax Tree) généré par le compilateur de notre langage Draw++. Chaque nœud de l'AST représente une structure syntaxique du programme, comme les affectations, les conditions ou les expressions binaires. Par exemple :

- Le nœud `Assignment` indique une opération d'affectation où une variable est associée à une valeur.
- Le nœud `IfStatement` représente une structure conditionnelle, avec des blocs pour les parties `then` et `else`.
- Les expressions, telles que les comparaisons (`a == 5`), sont encapsulées dans des nœuds `BinaryExpression`.

Cette représentation est essentielle pour permettre l'exécution ou la compilation du programme, en traduisant le code source en une structure exploitable par la machine.

Figures géométriques avec cercle

L'image ci-dessous illustre des figures géométriques générées à l'aide de commandes écrites dans le langage Draw++. Elle comprend :

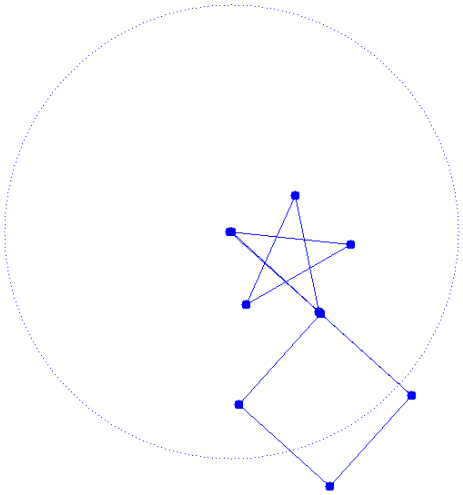


Figure 1: Figures géométriques générées avec Draw++

```
drawCursor(x, y)

startX -> x
startY -> y

pour i de 0 à 4  {
    moveCursor(dx, dy)
    drawLine(startX, startY, x, y)

    startX -> x
    startY -> y

    drawCursor(x, y)

    rotateCursor(90)
}

rotateCursor(90)
moveCursor(dx, dy)
drawLine(startX, startY, x, y)
drawCursor(x, y)

startX -> x
startY -> y

pour i de 0 à 5 {
    moveCursor(dx, dy)
    drawLine(startX, startY, x, y)

    startX -> x
    startY -> y

    drawCursor(x, y)

    rotateCursor(144)
}



angle -> 0
tantque angle < 360 {
    drawArc(x, y, 250, angle, angle + 10)
    angle -> angle + 10
}
```

Figure 2: Code Draw++

- Une **étoile** composée de segments de lignes, construite à partir de coordonnées spécifiques définies dans le code.
- Un **carré** , dessiné en reliant quatre points formant un polygone.
- Un **cercle** en pointillés englobant ces deux figures, tracé à partir d'un point central et d'un rayon prédéfini.

Ces formes démontrent les capacités du langage Draw++ à dessiner des objets complexes en combinant des instructions simples comme la définition de points, de lignes et de courbes. L'utilisation du cercle en pointillés illustre également les styles de tracé personnalisables disponibles dans le langage.