Python project :
Conception et réalisation du Langage Draw++

Louèva Béranger, Charles Delpech, Clara Viot, Mathéo Arondeau
et Noor-Lisa Mohamed Jubeer

Janvier 2025

Table des matières

1 Résumé

Le projet Draw++ consiste à concevoir un langage de programmation innovant dédié au dessin et à l'animation graphique. Il offre des instructions élémentaires, comme la création de curseurs, le dessin de formes, et la gestion des couleurs, ainsi que des instructions évoluées, telles que les boucles, les conditions et les regroupements d'instructions.

Pour accompagner ce langage, un IDE intégré est développé en python, permettant aux utilisateurs de créer, modifier, exécuter et déboguer leurs programmes facilement. Un compilateur, écrit en Python, est également conçu pour détecter les erreurs syntaxiques et générer un code intermédiaire en langage C.

2 Introduction

Dans le cadre de ce projet, nous avons entrepris de concevoir Draw++, un langage de programmation dédié à la création graphique et à l'animation. Ce langage permet aux utilisateurs de dessiner des formes, d'animer des objets et d'interagir avec des curseurs via un ensemble d'instructions simples et évoluées. L'objectif principal est de développer une plateforme intuitive pour créer des dessins programmés tout en renforçant les compétences des étudiants en conception de langages, programmation et compilation.

Le projet se décompose en plusieurs étapes clés : la définition d'une grammaire précise pour le langage, la conception d'un compilateur en Python capable de détecter les erreurs et de générer un code intermédiaire en langage C, ainsi que le développement d'un IDE interactif. Cette dernière offre des fonctionnalités comme l'édition, la sauvegarde, l'exécution et le débogage des programmes écrits en Draw++. Une documentation interactive accompagne également le projet pour guider les utilisateurs dans l'apprentissage et l'utilisation de ce nouveau langage.

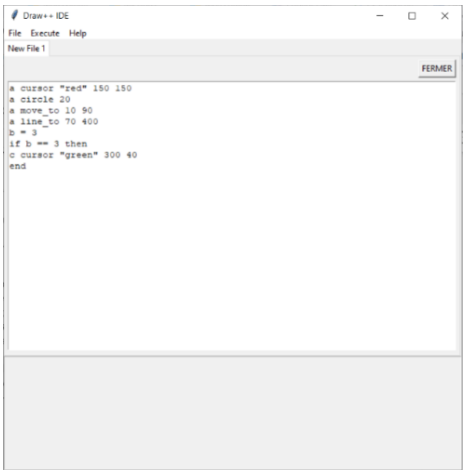

FIGURE 1 – Image du IDE

3 Description du projet

Le projet Draw++ vise à concevoir un langage de programmation nommé dédié à la création graphique et à l'animation, avec des fonctionnalités allant des commandes élémentaires aux instructions complexes. Ce projet est structuré autour de plusieurs axes : la définition des instructions, la conception de la grammaire, le développement d'un IDE afin de coder, et la réalisation d'un compilateur capable de générer un code intermédiaire en langage C afin de l'exécuter.

Avec notre projet, à partir de l'IDE il est possible d'écrire du code Draw++, d'ouvrir un ou plusieurs fichiers, de basculer d'un fichier à un autre, d'enregistrer un fichier, de l'exécuter. En dessous, il y a un espace pour écrire si la compilation s'est bien passée, s'il y a des erreurs de syntaxe ... Grâce au langage Draw++, on peut créer des curseurs d'une couleur spécifique à un emplacement spécifique sur une fenêtre de rendu. Nous pouvons le déplacer. Nous pouvons créer des cercles et des lignes à partir d'un curseur. Nous pouvons également mettre des "if" afin d'exécuter du code en fonction des conditions choisies. Sur la fenêtre de rendu, il est possible de sélectionner un objet, de le déplacer, de le supprimer, et de l'animer. Il est possible de créer plusieurs curseurs. Ci-dessus voici un aperçu de notre IDE, avec un exemple de code Draw++.

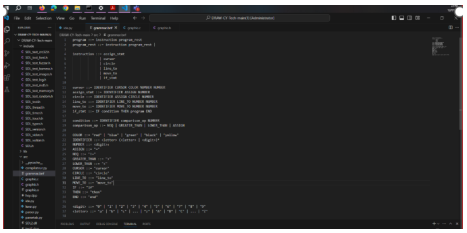

FIGURE 2 – Image de Grammaire

4 Grammaire et syntaxe de Draw ++

La grammaire est un élément fondamental qui définit la syntaxe et la structure des programmes écrits en Draw++. Elle garantit que les instructions respectent des règles bien définies, facilitant ainsi leur analyse, leur compilation et leur exécution. La grammaire de Draw++ s'appuie sur des commandes intuitives permettant de manipuler des curseurs et de dessiner des formes géométriques de manière dynamique. Voici notre grammaire :

Program : Un programme est constitué d'une ou plusieurs instructions. Cela peut inclure des affectations (`assign_stmt`), des déclarations de curseur (`cursor`), des cercles (`circle`), des déplacements de ligne (`line_to`), et des déplacements de curseur (`move_to`).

Instruction : Une instruction est soit une affectation, soit une commande de curseur, de cercle, de ligne ou de déplacement.

La commande `cursor` permet de définir un curseur avec sa couleur et sa position, tandis que `assign_stmt` permet d'affecter une valeur à une variable. La commande `circle` permet de dessiner un cercle, `line_to` permet de tracer une ligne, et `move_to` déplace le curseur à une nouvelle position.

Program_rest : Cette partie de la grammaire permet d'avoir plusieurs instructions consécutives dans un programme, mais aussi de définir la fin du programme (la règle vide).

Détails des commandes :

Cursor : Permet de définir un curseur avec un nom d'identifiant (`IDENTIFIER`), une couleur (`COLOR`), et des coordonnées (`NUMBER`).

Assign_stmt : Permet d'affecter une valeur numérique à une variable (`IDENTIFIER`).

Circle : Dessine un cercle en attribuant un rayon à une variable (`IDENTIFIER`).

Line_to et **Move_to** : Permettent de déplacer un curseur à des coordonnées spécifiques.

Définitions supplémentaires :

COLOR : Les couleurs autorisées pour un curseur.

IDENTIFIER : Un identifiant qui peut commencer par une lettre et être suivi de lettres ou de chiffres.

NUMBER : Un nombre entier (composé uniquement de chiffres).

ASSIGN : L'opérateur d'affectation (=).

NEQ : L'opérateur "différent de" (!=)

`GREATER_THAN`, `LOWER_THAN` : Les opérateurs de comparaison ("¡", "¿")

`CURSOR`, `CIRCLE`, `LINE_TO`, `MOVE_TO` : Des mots-clés du langage qui indiquent des actions spécifiques sur le curseur et les objets graphiques.

Pour résumer, voici la syntaxe précise avec des exemples :

- Pour déclarer une variable nommée "a" (uniquement des entiers) : `a = 2`
- Pour déclarer un curseur nommé "a" à la position (`x, y`) : `a cursor "red" x y`
- Pour déplacer un curseur : `a move_to x y`
- Pour tracer une ligne entre le curseur et un point précis à la position (`x, y`) : `a line_to x y`
- Pour dessiner un rond de rayon `x` à partir d'un curseur "a" : `a circle x`
- Pour mettre des conditions :
  - `if ... then`
  - `...`
  - `end`

5 Conception et l'IDE

Afin que l'utilisateur puisse utiliser le langage `Draw++`, il fallait coder une interface graphique permettant d'écrire du code.

1) Interface utilisateur avec `Tkinter`

L'interface utilisateur est créée à l'aide de `Tkinter`, une bibliothèque intégrée à Python pour développer des applications graphiques. Les principaux éléments incluent :

Fenêtre principale : Créée avec tk.Tk(), elle sert de conteneur global pour l'IDE.

Menu principal : Un menu interactif avec des options telles que File, Execute, et Help. Chaque menu propose des actions spécifiques, comme créer un nouveau fichier, ouvrir un fichier existant, enregistrer, compiler, ou afficher un fichier README.

Gestion des onglets : Le module ttk.Notebook est utilisé pour permettre l'ouverture de plusieurs fichiers simultanément dans des onglets distincts. Chaque onglet est un conteneur (tk.Frame) avec un éditeur de texte intégré.

Zone de sortie : Une zone dédiée (tk.Text) affiche les messages de compilation, d'exécution, ou les erreurs détectées dans le code.

2) Gestion des fichiers

Les fonctionnalités liées aux fichiers sont gérées à travers les menus et des boîtes de dialogue :

Création de nouveaux fichiers : Un onglet vierge est ajouté au Notebook avec un titre par défaut (ici New File 1).

Ouverture de fichiers : Avec filedialog.askopenfilename(), l'utilisateur peut ouvrir un fichier .dpp, dont le contenu est chargé dans un nouvel onglet.

Enregistrement des fichiers : filedialog.asksaveasfilename() permet d'enregistrer le contenu d'un onglet sous forme de fichier .dpp.

6 Lexer

Quand on écrit du code dans l'IDE, et qu'on appuie sur "execute", la fonction 'compile draw code' du fichier ide.py se lance. Plusieurs choses vont se passer. Tout d'abord, cette fonction récupère tout le code écrit dans l'IDE. Puis, la fonction crée une instance de la classe Lexer, et renvoie une liste de tous les tokens extraits du code source. Plus précisément, le lexer de notre projet Draw++ est chargé d'analyser le code source écrit dans le langage dédié et de le décomposer en une série de tokens. Ces tokens sont des unités lexicales, comme des mots-clés, des identifiants, des nombres ou des opérateurs, qui seront ensuite utilisés par le parser pour construire l'arbre syntaxique. Nous avons implémenté notre propre lexer sans l'aide d'aucun framework.

Le lexer fonctionne en parcourant le code source caractère par caractère. À chaque étape, il essaie de faire correspondre une portion du code avec un ensemble de modèles prédéfinis, spécifiés dans une liste appelée `token-specifications`. Chaque modèle associe un type de token (par exemple, `NUMBER`, `CURSOR`, `MOVE_TO`) à une expression régulière décrivant son format. Par exemple, les nombres sont capturés avec `\d+(\.\d+)?` pour inclure les entiers et les nombres à virgule flottante, tandis que les mots-clés comme `cursor` ou `move_to` ont leurs propres motifs.

Lorsqu'une correspondance est trouvée, le lexer vérifie si le token doit être ignoré (comme les espaces ou les retours à la ligne, représentés par le type SKIP) ou ajouté à la liste des tokens. Chaque token valide est enrichi avec des informations sur sa position, incluant la ligne et la colonne, ce qui est essentiel pour repérer les erreurs dans le code. Si aucun modèle ne correspond à un caractère donné, une erreur est levée, indiquant un caractère illégal avec sa position exacte.

Le lexer est donc une étape essentielle qui convertit le code brut en une structure organisée et compréhensible pour les étapes ultérieures de compilation, tout en offrant une gestion précise des erreurs lexicales.

Ci-dessous notre liste de tokens valides. L'ordre de ces derniers est important. Par exemple, si j'écris dans l'IDE a = 1, les tokens extraits de ce code seront :

('IDENTIFIER', 'a', 1, 1)

('ASSIGN', '=', 1, 3)

('NUMBER', '1', 1, 5)

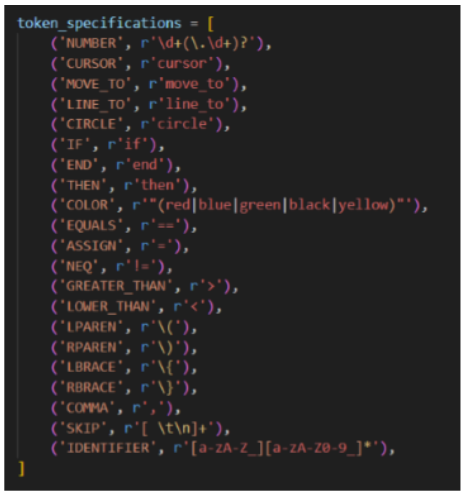

FIGURE 3 – Notre Lexer

7 Parser

L'étape de *parsing* (ou analyse syntaxique) suit l'analyse lexicale. Son objectif est de vérifier si la séquence de *tokens* produite par le *lexer* respecte la grammaire du langage. En d'autres termes, le parseur structure le code source sous forme d'arbre syntaxique abstrait (AST), qui représente la logique et la hiérarchie des instructions.

Dans la fonction `compile_draw_code` du fichier `ide.py`, une fois que le *lexer* retourne les *tokens* extraits du code source, la fonction crée une instance de `parser` qui va tenter d'analyser les *tokens* extraits.

Le Parser est une étape clé dans le processus de compilation ou d'interprétation du langage. Il prend les tokens générés par le Lexer et essaie de les organiser selon la grammaire du langage. Le Parser tente de comprendre la structure du programme et de construire une représentation interne, généralement sous forme d'un arbre syntaxique (ou Abstract Syntax Tree, AST). Un AST est une représentation arborescente d'un programme source qui capture la structure hiérarchique des instructions d'un langage de programmation. Chaque nœud de l'AST représente une construction syntaxique ou un élément clé du programme, tel qu'une opération, une affectation, ou une déclaration. Ces nœuds sont reliés entre eux par des branches, formant une structure d'arbre. Un nœud peut avoir des enfants, qui correspondent à des éléments syntaxiques plus spécifiques ou des sous-expressions. Par exemple, dans notre projet, un nœud représentant une affectation dans un programme source aura un type "assign_stmt" et des enfants représentant la variable à laquelle on assigne une valeur et la valeur elle-même.

Ci-dessous, un extrait de la fonction `compile_draw_code`, où la méthode

FIGURE 4 – Détermination de la Grammaire pour 'Cursor'

`parse()` du `Parser` prend la liste des *tokens* comme entrée et les examine pour construire un arbre syntaxique (*AST*).

La méthode parse() du Parser prend la liste des tokens comme entrée et les examine pour construire un arbre syntaxique (AST). L'AST est une représentation arborescente du programme source, qui montre la hiérarchie et la relation entre les différents éléments du programme.

Pendant l'analyse, le Parser utilise la grammaire du langage définie dans le fichier parser.py pour déterminer si la séquence de tokens forme des constructions valides du langage. Si une séquence de tokens ne suit pas la syntaxe attendue, le Parser générera des erreurs.

Si l'analyse est réussie, la méthode parse() retourne une structure représentant le programme sous forme d'instructions ou de nœuds de l'AST.

Si une erreur de syntaxe est détectée, une exception est généralement levée, indiquant l'emplacement de l'erreur dans le code source, et un message d'erreur est généré. Ci-dessous, un exemple de code qui détermine la grammaire de l'instruction "cursor" (pour créer un curseur). Cette partie du code va vérifier si les tokens extraits respectent la syntaxe définie ci-dessous, et va générer une instruction.

Grâce à ceci, si j'écris : `a cursor "red" 150 150`, l'AST et les tokens retournés seront :

```
('IDENTIFIER', 'a', 1, 1)
('CURSOR', 'cursor', 1, 3)
('COLOR', '"red"', 1, 10)
('NUMBER', '80', 1, 16)
('NUMBER', '80', 1, 19)
```

Et à partir de ces tokens, le parseur retournera :

```
('program', [('cursor_stmt', ['a', '"red"', '80', '80']), ('program_rest', [])])
```

Pourquoi va-t-il retourner ceci ? C'est grâce à cette partie du code (provenant du fichier `parser.py`) :

Par exemple, si le premier token analysé est `IDENTIFIER`, suivi d'un token `CURSOR`, le programme va aller dans la fonction `cursor_stmt`, vérifier si la syntaxe correspond (`IDENTIFIER`, `CURSOR`, `COLOR`, `NUMBER`, `NUMBER`). Si c'est le cas, le nœud correspondant sera retourné.

La liste `instructions` de la fonction `compile_draw_code` sera remplie par l'AST généré par les tokens extraits.

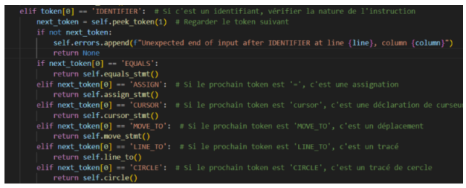

FIGURE 5 – Partie du fichier `parser.py`

FIGURE 6 – Cursor

8 Traduction en C

Ensuite, s'il n'y a aucune erreur de syntaxe dans le code fourni, la fonction passe à la traduction en C. La traduction en C permet de convertir l'AST (Arbre Syntaxique Abstrait) retourné par le parser en un code exécutable dans un environnement compatible avec le langage C.

Cette étape est importante pour exécuter le code `Draw++`, car le programme C généré peut être compilé et exécuté directement. Elle optimise également les performances, car le langage C est bien adapté aux tâches graphiques.

Ici, l'AST retourné par le parser (`instructions`) est passé dans la fonction `ast_to_c`, qui va le traduire en un code C exécutable. Par exemple, si la fonction reçoit un AST tel que celui-ci :

```
('program', [('assign_stmt', ['a', '1']), ('program_rest', [])])
```

La fonction retournera :

```
int a = 1;
```

La fonction `ast_to_c` analyse récursivement les nœuds de l'AST. Les nœuds correspondent à chaque instruction du programme.

Selon le type du nœud (déterminé par `node_type`), elle génère des

FIGURE 7 – Traduction de l'AST en un code C excécutable

morceaux spécifiques de code C.

Le code C généré sera copié dans un fichier C ayant pour nom le nom de l'onglet de l'IDE dans lequel nous sommes en train d'écrire. Ce fichier C sera notre exécutable.

FIGURE 8 –

9 Exécution du code C

Dans cette section du programme, nous gérons la compilation et l'exécution d'un programme C généré à partir du code source écrit dans notre langage `Draw++`. Lors de la compilation, nous utilisons le compilateur `gcc` pour transformer le code source en un exécutable C. La commande de compilation spécifie l'emplacement des bibliothèques nécessaires, y compris la bibliothèque `graphic`, que nous avons créée nous-mêmes pour gérer les fonctionnalités graphiques du programme. Il est donc essentiel de compiler cette bibliothèque avant d'exécuter le programme, car elle contient des fonctions indispensables au bon fonctionnement du code généré, telles que la gestion des curseurs, des formes géométriques, et des événements graphiques.

En plus de cette bibliothèque personnalisée, la commande inclut également la bibliothèque `SDL2`, utilisée pour la gestion des événements et de l'affichage graphique. Il est impératif que `SDL2` soit installé sur la machine et correctement paramétré pour que la compilation se déroule sans erreur. Cela inclut le bon emplacement des fichiers d'en-tête et des bibliothèques lors de la compilation. Si ces fichiers ne sont pas trouvés ou si l'installation de `SDL2` est incorrecte, la compilation échouera, et un message d'erreur sera affiché.

Ci-dessous, un exemple de dessin pouvant être réalisés avec notre projet :
Grâce à notre bibliothèque personnelle utilisant SDL2, sur la fenêtre de rendu il est possible de sélectionner avec le clic gauche en glissant un ou plusieurs objets, de les déplacer (toujours avec le clic gauche), en faisant un clic droit sur un objet sélectionné il est possible de l'animer. Notre animation est la suivante : l'objet va se déplacer en faisant un carré, 3 fois. Il est également possible de supprimer un objet de la fenêtre : il suffit de le sélectionner, et d'appuyer sur la touche "Entrée" du clavier.

Ci-dessous un exemple de fonction C de notre bibliothèque nommée

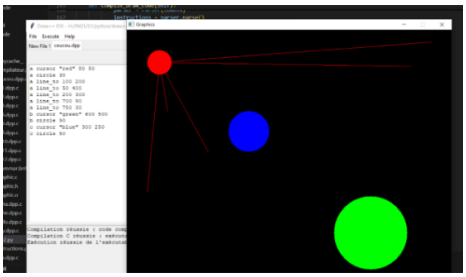

FIGURE 9 – Résultat d'excécution d'un programme Draw++

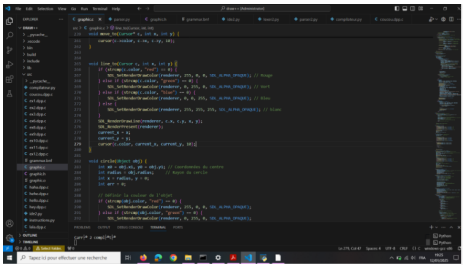

FIGURE 10 – Image du code graphic.c

`graphic` qui permet de dessiner une ligne allant du curseur jusqu'à une position (opérande de la fonction `line_to`) :

10 Gestion des erreurs

Notre IDE permet de détecter les erreurs de syntaxes faites par le programmeur. Ainsi, les lignes où il y a des erreurs de syntaxes seront surlignées en jaune, et le texte sera écrit en rouge. Dans la fenêtre en bas, un message s'affichera nous disant à quelle ligne se situe le ou les erreurs de syntaxes. La gestion des erreurs est centralisée dans le processus d'analyse et de parsing. Lorsque le parser rencontre un problème, comme une erreur de syntaxe ou un token inattendu, il enregistre l'erreur dans une liste d'erreurs. Ce processus commence par l'analyse de chaque token et se poursuit au sein de diverses méthodes d'analyse spécifiques (comme `cursor_stmt()`, `move_stmt()`, etc.). Si un token ne correspond pas à ce qui est attendu ou si le programme rencontre un état inattendu, une erreur est ajoutée à la liste `self.errors`. Chaque erreur contient des informations détaillées, telles que le type d'erreur, la ligne et la colonne où elle s'est produite, pour faciliter le débogage et permettre à l'utilisateur d'identifier précisément où se situe le problème dans le code source.

Lorsque les erreurs sont récoltées dans le processus de parsing, elles sont d'abord capturées dans la méthode `instruction()`. Par exemple, lorsqu'un token inattendu est rencontré, le parser ajoute un message d'erreur, avec des informations sur la ligne et la colonne du token problématique, à la liste des erreurs.

FIGURE 11 – Extrait du fichier parse.py, permet d'enregistrer les erreurs dans une liste

Cela permet d'accumuler toutes les erreurs pendant l'analyse du code source. Une fois toutes les erreurs détectées, elles sont traitées dans un autre bloc de code qui surligne les erreurs dans l'interface graphique, ce qui permet à l'utilisateur de voir immédiatement où se trouvent les erreurs dans le code.

Ce code montre comment, en cas de token inattendu ou de situation

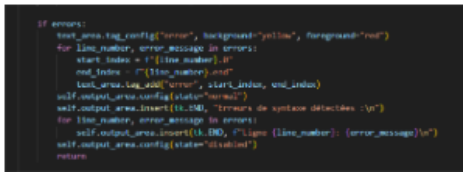

FIGURE 12 – Extrait du code dans compile$_{d}raw_{c}odequivasurlignerenjauneleslignesoudeserreurssonttrouvées$

imprévue, des erreurs sont immédiatement enregistrées avec des informations détaillées.

Ainsi, les erreurs avec leur numéro de ligne sont récupérées dans `compile_draw_code`. Si des erreurs sont détectées, l'interface graphique est mise à jour. Les lignes contenant des erreurs sont surlignées dans la fenêtre d'édition à l'aide d'un surlignage jaune avec un texte en rouge, afin d'attirer l'attention de l'utilisateur sur les parties problématiques du code.

Une fois cette mise en forme effectuée, un message d'erreur détaillé est affiché dans la zone de sortie, précisant la ligne et le message d'erreur pour chaque problème rencontré. Cette approche permet à l'utilisateur d'identifier rapidement et de corriger les erreurs dans son code, améliorant ainsi l'expérience de développement dans notre environnement intégré.

11 Conclusion

En conclusion, notre projet avait pour but de créer un langage de programmation graphique, Draw++, permettant de dessiner et manipuler des objets à l'écran via un code simple et intuitif. Nous avons conçu la grammaire du langage, développé un lexer et un parser pour l'interpréter, puis traduit le code source en langage C. Le code C généré est ensuite compilé et exécuté, affichant les résultats sur une fenêtre graphique utilisant la bibliothèque SDL2. Une bibliothèque personnalisée, graphic, a été créée pour gérer les objets graphiques, les déplacements, et les interactions avec l'utilisateur.

Tout au long du projet, nous avons travaillé à garantir que le système fonctionne de manière fluide et que le processus de compilation et d'exécution soit simple et direct. Bien que des fonctionnalités telles que l'ajout de boucles, la rotation des objets ou encore un mécanisme de zoom aient été envisagées, nous avons fait le choix de privilégier un code stable et fonctionnel, respectant les exigences du projet dans le temps imparti. Nous avons ainsi réussi à fournir une solution qui répond aux objectifs principaux du projet tout en restant ouverte à de futures améliorations et ajouts de fonctionnalités.

Ce projet aura été un défi a plusieurs niveaux pour nous à travers les mois. Une utilisation de d'Internet constante pour réussir à avancer sur le projet et une répartition des tâches complexes. Nous sommes tout de même fiers de présenter notre projet, et de toutes les choses que nous avons appris sur le chemin.