Rapport – Compilation 1

TP Compilation : Construction d'un Analyseur Syntaxique pour un mini Langage C

REALISE PAR :

DIALLO Fatimatou

DIALLO Mamadou Talibe

Le, 19/12/2025

ENCADRE PAR :

M. Karim tamine, M. Tristan Vaccon

Faculté des Sciences et Techniques de Limoges

1) Introduction

Dans le cadre de notre projet de **compilation**, notre objectif principal était de **développer un analyseur syntaxique** capable de **prendre en entrée une chaîne représentant un petit programme** et de **retourner l'arbre syntaxique** correspondant aux **règles de la grammaire** définie. Avant de passer à la partie pratique, nous avons d'abord **vérifié en papier que notre grammaire était bien une grammaire LL(1)**, ce qui est essentiel pour pouvoir construire un analyseur descendant prédictif.

Après avoir conçu une **première version en ligne de commande**, nous avons voulu rendre notre travail **plus interactif et agréable à utiliser**. Pour cela, nous avons développé une **version graphique** en utilisant **Tkinter**, une bibliothèque Python dédiée à la création d'interfaces. Cette version permet à l'utilisateur de **saisir une chaîne**, de **l'analyser facilement**, puis d'**afficher les règles appliquées** ainsi que **l'arbre syntaxique généré**. En plus d'améliorer la présentation, cette approche nous a aidés à **mieux comprendre le lien entre la logique d'analyse et l'affichage graphique**, tout en rendant notre projet plus **pratique et visuel**.

Enfin, dans ce rapport, nous expliquerons la **démarche que nous avons suivie** ainsi que les **choix que nous avons faits** afin de **mener à bien ce travail**, depuis la conception de l'analyseur jusqu'à la réalisation de l'interface graphique avec Tkinter.

2) Grammaire du langage

Voici la grammaire étudiée pour ce projet :

<Programme> ::= main() {<liste_declarations><liste_instructions>}

<liste_declarations> ::= <une_declaration><liste_declarations> | vide

<une_declaration> ::= <type>id

<liste_instructions> ::= <une_instruction><liste_instructions> | vide

<une_instruction> ::= <affectation>|<test>

<type> ::= int | float

<affectation> ::= id=nombre;

<test> ::= if <condition> <une_instruction> else <une_instruction>

<condition> ::= id<operateur>nombre

<operateur> ::= < | > | ==

Les symboles **terminaux** sont les éléments finaux du langage que l'analyseur lit : **main(){**, **if**, **else**, **;** , **}**, **id**, **nombre** , **int**, **float**, **<**, **>**, **=**, **==**.

Les symboles **non-terminaux** représentent les catégories syntaxiques définies par la grammaire : <**Programme**>, <**liste_declarations**>, <**une_declaration**>, <**liste_instructions**>, <**une_instruction**>, <**type**>, <**affectation**>, <**test**>, <**condition**> et <**operateur**>.

3)Définitions de quelques termes

a- Symboles : **un symbole** représente un **élément de la grammaire**, Il peut être :

- **non terminal** (comme <Programme>, <liste_instructions>, <condition>, etc.)
- ou **terminal** (comme id, if, nombre, ;, etc.)

b- Règles :

En compilation, une **règle** (ou **règle de production**) est une **instruction grammaticale** qui décrit **comment construire ou reconnaître une structure du langage** à partir de symboles plus simples. Elle fait partie de la **grammaire** du langage, utilisée par l'analyseur syntaxique pour **vérifier la structure d'un programme** et **construire l'arbre syntaxique**.

c- Table d'analyse LL(1) :

La **table d'analyse LL(1)** est une structure utilisée par un **analyseur syntaxique descendant prédictif** pour déterminer automatiquement quelle règle appliquer à chaque étape de l'analyse. Elle permet de **guider la dérivation d'une phrase** tout en évitant les ambiguïtés et en vérifiant que le programme respecte la grammaire.

Cette table est construite à partir des ensembles **PREMIER** et **SUIVANT** :

- Les **PREMIER** permettent de savoir quels symboles peuvent commencer une dérivation pour chaque non-terminal.
- Les **SUIVANT** indiquent quels symboles peuvent apparaître après un non-terminal, notamment lorsqu'il peut produire le vide ε.

En combinant ces informations, on peut remplir la table LL(1) : chaque case indique la **règle de production à appliquer** selon le symbole d'entrée courant et le non-terminal associé.

4) Grammaire LL(1) : Calcul des ensembles PREMIER et SUIVANT

1. <Programme>

- PREMIER(main() { <liste_declarations> <liste_instructions> }) = { **main(){** }
- Vérification : une seule règle → pas de conflit

2. <Liste_declarations>

- Règle 1 : PREMIER(<une_declaration> <liste_declarations>) = { **int, float** }
- PREMIER(vide) = { **vide** }, SUIVANT(<liste_declarations>) = { **if, id, }** }
- Vérification : Règle 1 ∩ SUIVANT(<liste_instructions>) = **∅** → pas de conflit

3. <Une_declaration>

- PREMIER(<type> id) = { **int, float** }
- Vérification : une seule règle → pas de conflit

4. <Liste_instructions>

- Règle 1 : PREMIER(<une_instruction> <liste_instructions>) = { **id, if** }
- PREMIER(vide) = { **vide** }, SUIVANT(<liste_instructions>) = { **}, vide** }
- Vérification : Règle1 ∩ SUIVANT(<liste_instructions>) = **∅** → pas de conflit

5. <Une_instruction>

- Règle 1 : PREMIER(<affectation>) = { **id** }
- Règle 2 : PREMIER(<test>) = { **if** }
- Vérification : Règle1 ∩ Règle2 = **∅** → pas de conflit

6. <Type>

- Règle 1 : PREMIER(int) = { **int** }
- Règle 2 : PREMIER(float) = { **float** }
- Vérification : Règle1 ∩ Règle2 = **∅** → pas de conflit

7. <Affectation>

- Règle 1 : PREMIER(id = nombre;) = { **id** }
- Vérification : une seule règle → pas de conflit

8. <Test>

- Règle 1 : PREMIER(if <condition> <instruction> else <instruction>) = { **if** }
- Vérification : une seule règle → pas de conflit

9. <Condition>

- Règle 1 : PREMIER(id <operateur> nombre) = { **id** }
- Vérification : une seule règle → pas de conflit

10. <Operateur>

- Règle 1 : PREMIER(<) = { **<** }
- Règle 2 : PREMIER(>) = { **>** }
- Règle 3 : PREMIER(==) = { **==** }
- Vérification : Règle1 ∩ Règle2 = **∅**, Règle1 ∩ Règle3 = **∅**, Règle2 ∩ Règle3 = **∅** → pas de conflit

Après vérification de **tous les non-terminaux**, aucun conflit n'est constaté dans les ensembles PREMIER et SUIVANT.**On peut donc conclure que la grammaire est LL(1)** et peut être utilisée pour construire la table d'analyse LL(1).

5)Table d'analyse

Pour rendre la table d'analyse plus lisible, nous avons utilisé la numérotation des règles vue en classe, afin de faciliter la compréhension.

1. <Programme> ::= main() {<liste_declarations><liste_instructions>}
2. <liste_declarations> ::= <une_declaration><liste_declarations>

3. <liste_declarations> ::= vide
4. <une_declaration> ::= <type>id
5. <liste_instructions> ::= <une_instruction><liste_instructions>
6. <liste_instructions> ::= vide
7. <une_instruction> ::= <affectation>
8. <une_instruction> ::= <test>
9. <type> ::= int
10. <type> ::= float
11. <affectation> ::= id=nombre;
12. <test> ::= if <condition> <instruction> else <instruction>
13. <condition> ::= id<operateur>nombre
14. <operateur> ::= <
15. <operateur> ::= >
16. <operateur> ::= ==

	main(){	}	int	float	id	nombre	=	<	>	==	if	else	;	vide
<Programme>	1													
<Liste_declarations>		3	2	2	3						3			3
<Une_declaration>			4	4										
<Liste_instructions>		6			5						5			6
<Une_instruction>					7						8			
<Type>			9	10										
<Affectation>					11									
<Test>											12			
<Condition>					13									
<Operation>								14	15	16				

6) Explication du code

1) Analyseur.py:

Dans ce fichier, nous avons défini la **grammaire** de notre langage sous forme de **table d'analyse syntaxique** (tableAnalyse) via un **dictionnaire de données**. Cette table permet de guider l'analyse descendante en associant chaque paire (non-terminal, terminal) à une règle de production.

Nous avons également implémenté une fonction appelée analyser_chaine(), qui repose sur l'algorithme vu en cours. Cette fonction permet de déterminer si une chaîne appartient au langage généré par notre grammaire. Elle utilise une **pile** pour simuler le processus d'analyse syntaxique, en appliquant les règles de la table selon les symboles rencontrés.

La fonction decouper_chaine() permet de découper la chaîne d'entrée en **tokens** reconnus par notre analyseur, en tenant compte des symboles complexes comme main(){.

2) Arbre.py :

Ce fichier permet de **construire et d'afficher l'arbre syntaxique** généré à partir de l'analyse grammaticale d'une chaîne. Il contient plusieurs parties :

1. **Classe Node** : représente un nœud de l'arbre. Chaque nœud contient :
   - value : le symbole qu'il représente.
   - children : la liste de ses enfants.

- nb_feuilles : le nombre de feuilles sous ce nœud, utilisé pour placer les nœuds horizontalement.
  La méthode add_child ajoute un enfant à un nœud.

2. **construire_arbre(liste_regle)** : crée l'arbre à partir des règles de l'analyse.
   - Le symbole gauche devient un nœud parent.
   - Les symboles à droite deviennent des enfants.
   - Si un enfant est un symbole gauche d'une autre règle, il est relié correctement.
   - Retourne la racine de l'arbre.
3. **compter_feuilles(node)** : calcule le nombre de feuilles sous chaque nœud.
   - Si le nœud n'a pas d'enfants, sa valeur est 1.
   - Sinon, c'est la somme des feuilles de ses enfants.
   - Sert à répartir les nœuds horizontalement sans chevauchement.
4. **placer_noeuds(noeud, profondeur, x_min, x_max, positions, hauteur, marge)** : calcule les positions (x, y) des nœuds.
   - Le nœud courant est centré horizontalement entre x_min et x_max.
   - La profondeur détermine la position verticale.
   - Les enfants sont placés proportionnellement au nombre de feuilles pour un arbre équilibré.
5. **ArbreGraphique** : dessine l'arbre dans un **Canvas Tkinter**.
   - afficher(racine) : initialise l'affichage.
   - redessiner() : efface et redessine l'arbre.
   - dessiner_depuis_positions(positions) : dessine les rectangles, le texte et les lignes parent-enfant.
   - zoom_molette(event) : permet de zoomer avec la molette de la souris.

Le Canvas utilise des scrollbars si l'arbre dépasse l'écran.

En résumé, ce code construit l'arbre à partir des règles, calcule les positions des nœuds et affiche un arbre **interactif, hiérarchique et navigable**.

3)fenetre.py :

Ce fichier est le **point d'entrée de l'interface graphique** de notre analyseur syntaxique. Il gère la **création, la configuration et la navigation** entre les différentes fenêtres Tkinter, elle comporte les fonctions suivantes :

lancer_analyse() :

Analyse la chaîne entrée par l'utilisateur.
Affiche un message d'erreur si la chaîne est vide ou non reconnue.
Passe à l'écran de succès si la chaîne est acceptée.

affiche_regles() :

- Affiche les règles appliquées pendant l'analyse,
- Utilise une TextBox pour les présenter proprement,
- Permet de revenir à l'écran précédent.

generer_arbre() :

- Construit l'arbre syntaxique à partir des règles.
- L'affiche sur un Canvas avec Tkinter.
- Gère la disposition des nœuds pour éviter les chevauchements.

7) Tests et Résultats

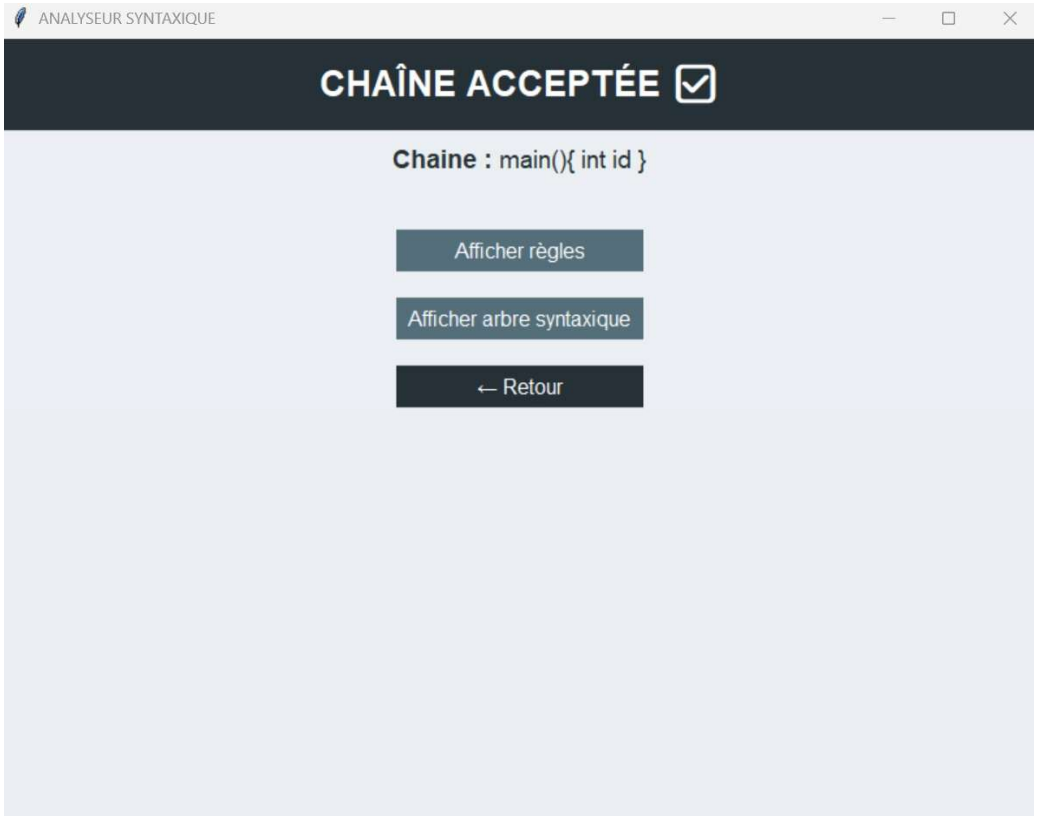
1) Affichage de la page d'accueil :

2) Affichage d'un message d'alerte si aucune chaîne n'est saisie lors du clic sur 'Analyser' :

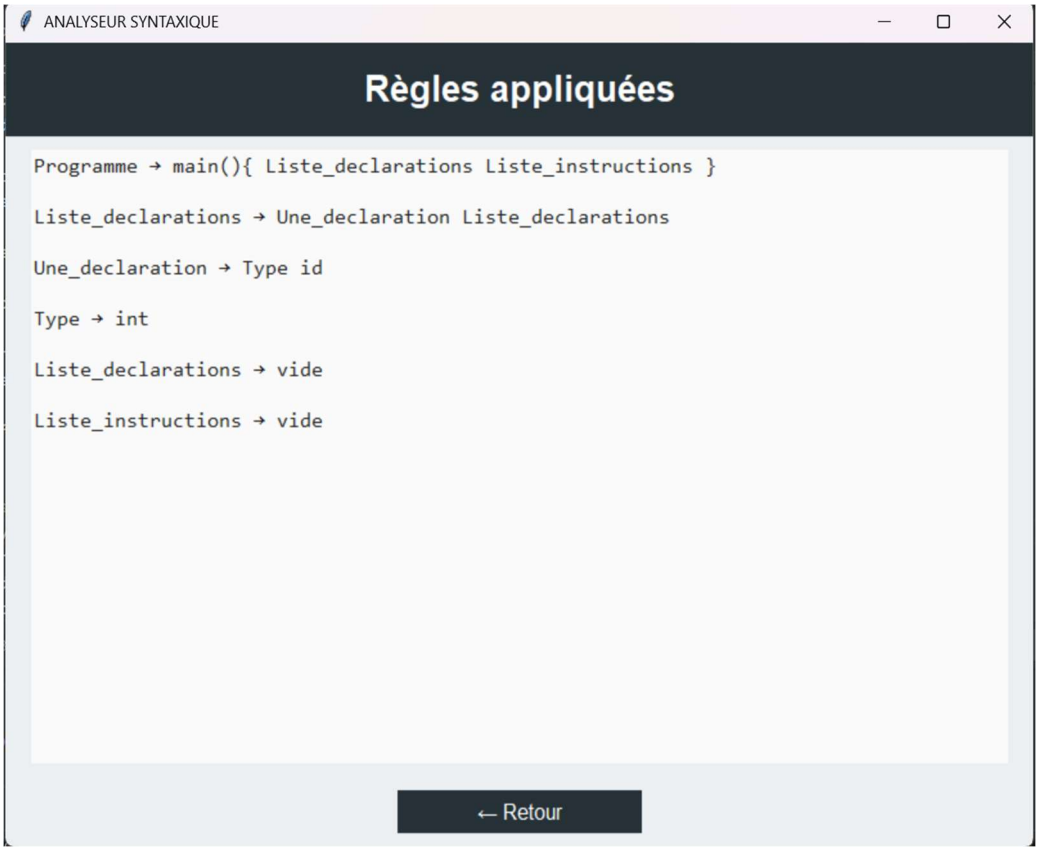
3) Tester une chaine qui n’est pas correcte

4) Donner une chaine correcte

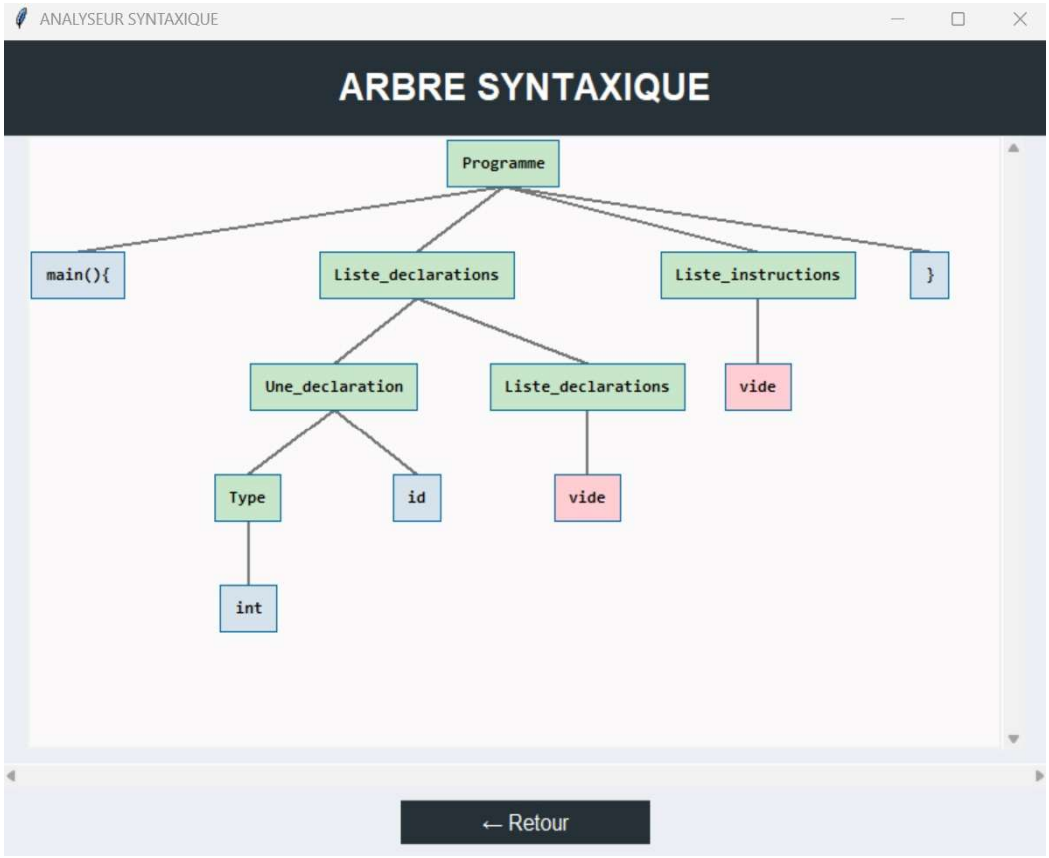
5) Passer à la fenêtre suivante (si le mot est accepté par la grammaire)



6) Affichage des règles appliquées :



7) Affichage de l'arbre



8) Conclusion

Pour réaliser ce projet, il a été essentiel de **préparer au préalable la table d'analyse**. Une fois cette étape effectuée, les difficultés rencontrées ont été beaucoup moins importantes.

L'un des principaux obstacles a été **l'affichage de l'arbre syntaxique**. En effet, certaines chaînes reconnues par notre langage, générées à partir de la grammaire, pouvaient être très longues et dépasser la zone d'affichage. C'est pour cette raison que l'ajout du **zoom** a été nécessaire, afin de rendre l'interface beaucoup plus intuitive et facile à manipuler.

En résumé, ce projet nous a permis de **mettre en pratique les notions étudiées en cours et en TD**. Il nous a également donné l'occasion de **revoir les concepts liés à Tkinter**, que nous avions utilisés l'année précédente dans le cadre de notre projet du semestre 4.