INFOB314 : Syntaxe et Sémantique des Langages

Rapport du projet En-Mode-J

Auteurs :
Groupe 14 :

Bruge TCHATCHOUA NDOUTENG
Loic DJUIDEU TCHOUAMOU
Loic FranckKAMGAIN MEUPIAP
Aubry MBIENDOU

May 25, 2025

Table des matières

1 Introduction

Dans le cadre du cours INFOB314, nous avons conçu un compilateur pour le langage de programmation **EMJ**, un langage à base d'*émojis* permettant de piloter le robot CuteBot. Développé en Java avec ANTLR, le compilateur prend en entrée un programme EMJ, effectue successivement l'analyse lexicale et syntaxique, la vérification sémantique (fondée sur une **table des symboles** et un patron *visiteur*) puis génère du code **MicroPython** exécutable sur le robot. Le présent rapport retrace les phases clés et les aspects techniques du projet : conception, organisation et implémentation du compilateur.

Après avoir décrit la structure des données employée — en particulier la **table des symboles**, indispensable pour la gestion des identificateurs durant la compilation — nous présenterons l'architecture globale du compilateur : fonctions de chaque composant et justifications de nos choix de conception.

Nous mettrons ensuite l'accent sur les classes essentielles du projet, notamment `EMJVisitor`, `EMJCodeGenerator`, `AllSymbol` et `TableSymbol`, en précisant leurs responsabilités respectives et la manière dont elles interagissent tout au long du processus de compilation.

Ce projet illustre notre maîtrise des principes théoriques et pratiques de la compilation et fournit une solution opérationnelle pour compiler et exécuter des programmes écrits en EMJ.

2 Structure de données et utilisation de la table des symboles

2.1 Description générale

La **table des symboles** est une structure de données cruciale dans un compilateur. Elle est utilisée pour stocker des informations sur les identifiants (noms de variables, fonctions, etc.) rencontrés dans le code source, et elle permet de valider la cohérence et la validité des utilisations de ces identifiants au cours de la compilation. Concrètement, la table des symboles permet de vérifier qu'une variable est déclarée avant d'être utilisée, d'empêcher les déclarations multiples d'un même identifiant dans une portée, de s'assurer qu'une fonction appelée existe et que le type de sa valeur de retour est correctement exploité, etc. Dans notre compilateur `EMJ`, la table des symboles est implémentée via deux classes principales : `AllSymbol` et `TableSymbol`. La classe `AllSymbol` définit les informations stockées pour chaque symbole, tandis que `TableSymbol` gère la collection de tous les symboles et fournit des méthodes pour ajouter ou rechercher des symboles en fonction de la portée courante.

2.2 Diagramme de classe

Le diagramme suivant(diponible à la page 4) illustre les principales classes de notre compilateur**EMJ**. Pour optimiser la lisibilité, il est pivoté d'un quart de tour ;tournez simplement la page (ou lisez la version PDF qui fait la rotation automatiquement).

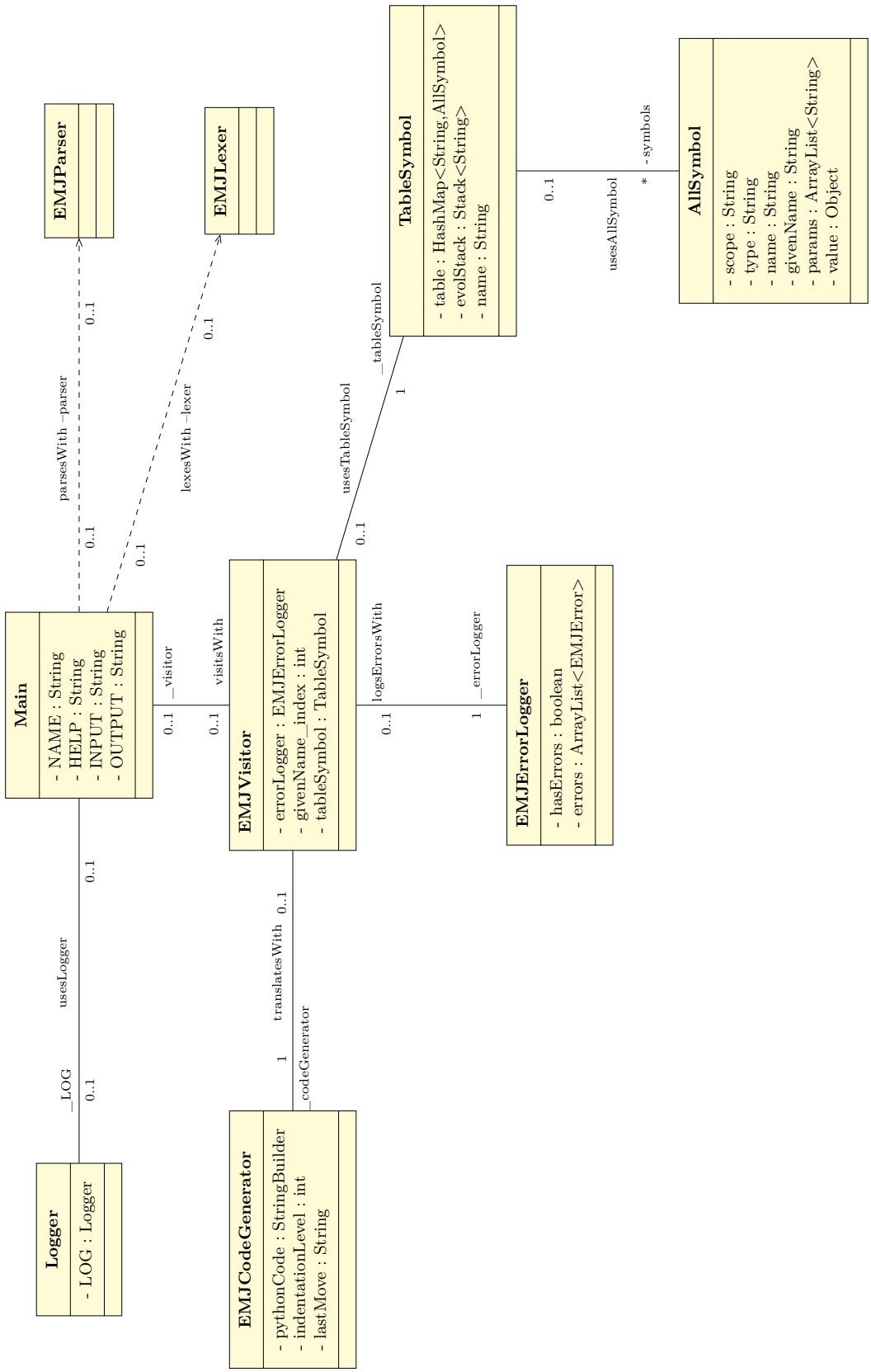

Figure 1 – Diagramme de classes complet du compilateur **EMJ**

3 Justification des choix de conception

Dans le cadre du développement du compilateur EMJ, plusieurs choix de conception ont été faits afin d'optimiser la gestion des symboles, d'améliorer l'efficacité du processus de compilation et d'assurer la robustesse du code généré. Nous détaillons ci-dessous les principaux choix et leurs justifications.

3.1 Utilisation de `HashMap` pour la table des symboles

La table des symboles est implémentée sous forme de `HashMap< String, AllSymbol >`, ce qui présente deux avantages :

- **Efficacité des accès** : insertion et recherche s'effectuent en temps quasi constant ($O(1)$), ce qui réduit le temps de compilation lorsque les identifiants sont nombreux.
- **Gestion dynamique** : les symboles peuvent être ajoutés (ou retirés) à la volée au fil de l'analyse sans impact notable sur les performances, ce qui simplifie la mise à jour de la table pendant le parcours de l'AST.

3.2 Utilisation de la pile (`Stack`) pour la gestion des portées

Le suivi des portées lexicales repose sur une `Stack<String>` :

- **Correspondance naturelle avec l'imbrication des blocs** : chaque entrée dans un nouveau bloc (*fonction*, *if*, *while*, ...) entraîne un `push`, et la sortie du bloc déclenche un `pop`.
- **Simplicité et clarté** : les méthodes `enterScope(scopeName)` et `exitScope()` suffisent à gérer l'évolution de la portée courante, rendant le code plus lisible.

3.3 Séparation des responsabilités

- **Analyse syntaxique** : assurée exclusivement par les classes générées par ANTLR4 (`EMJLexer`, `EMJParser`).
- **Analyse sémantique** : centralisée dans `EMJVisitor`, qui remplit la table des symboles et enregistre les erreurs dans `EMJErrorLogger`.
- **Génération de code** : déléguée à `EMJCodeGenerator`, totalement découplé du visiteur sémantique.
- **Gestion des erreurs** : un logger unique collecte les diagnostics, tandis que `Main` orchestre l'arrêt ou la poursuite de la compilation selon leur présence.

3.4 Gestion de l'indentation pour le code cible Python

Le compilateur maintient un compteur `indentationLevel` et recourt à une fonction utilitaire `appendLine()` :

- Chaque ligne Python est préfixée par `indentationLevel $\times$ 4` espaces, garantissant une indentation correcte des blocs `if`, `while`, `for`, etc.
- L'incrément/décrément du compteur est centralisé dans les méthodes `visitIfInstr`, `visitLoopInstr`, `visitFuncDecl`, ..., ce qui réduit les risques d'erreurs de mise en forme.

3.5 Variable `lastMove` pour le suivi du robot

- **Optimisation des rotations** : `lastMove` mémorise la dernière direction empruntée ; avant chaque nouveau déplacement, l'algorithme calcule la rotation minimale à effectuer.

4 Architecture générale

L'architecture du projet se décompose en plusieurs packages correspondant aux différentes phases du compilateur et à ses utilitaires. Nous distinguons principalement la partie syntaxique (grammaire et analyse syntaxique ANTLR) et la partie sémantique/génération (implantée en Java), sans oublier le support pour les tests. Ci-dessous, nous décrivons brièvement ces composantes et leur organisation :

4.1 Analyse syntaxique

La définition du langage EMJ est assurée par des grammaires ANTLR4, situées dans le dossier src/main/antlr4/be/unamur/info/b314/compiler. On y trouve deux fichiers : **EMJLexer.g4** (grammaire lexicale) et **EMJParser.g4** (grammaire syntaxique).

`be.unamur.info.b314.compiler` : ce package contient les classes générées par ANTLR à partir de la grammaire définie pour le langage EMJ (telles que `EnModeJParser` et `EnModeJLexer`). Il regroupe également les classes principales comme `Main` qui orchestre le processus global de compilation.

- `EMJLexer.g4` : la grammaire *lexicale* qui spécifie l'ensemble des *tokens* du langage (mots-clés, identifiants, littéraux, emojis, etc.). Elle gère également l'ignorance des commentaires et des espaces superflus.
- `EMJParser.g4` : la grammaire *syntaxique* qui décrit la structure complète d'un programme EMJ. La règle de départ `root` accepte soit une définition de carte (`map`), soit un programme complet comprenant les `imports` éventuels, la fonction `main` et les définitions de fonctions utilisateur.

Une fois ces grammaires définies, ANTLR4 génère automatiquement les classes Java correspondantes — le *lexer* `EMJLexer.java`, le *parser* `EMJParser.java`, ainsi que les stubs de visiteur `EMJParserBaseVisitor.java`, etc., stockés sous `target/generated-sources/antlr4`. Ces classes permettent de transformer un fichier source **EMJ** en un arbre syntaxique (*ParseTree*). L'utilisation d'ANTLR nous a ainsi fourni une base fiable pour la phase syntaxique, nous permettant de nous concentrer sur l'implémentation de la phase sémantique et de la génération de code.

4.2 Analyse sémantique

La logique *sémantique*, la *génération de code* et l'ensemble des utilitaires sont implémentés dans le répertoire `src/main/java/be/unamur/info/b314/compiler` ; l'architecture se répartit en plusieurs sous-packages cohérents :

- `compiler.main`
  - `Main` : point d'entrée du compilateur. Orchestration des phases *lexing*, *parsing*, *visite sémantique*, *génération MicroPython* et écriture du fichier `.py`, tout en affichant *OK/KO* selon la présence d'erreurs.
- `compiler.emj`
  - `EMJVisitor` : visiteur sémantique remplissant la table des symboles et enregistrant les erreurs via `EMJErrorLogger`.
  - `EMJCodeGenerator` : visiteur dédié à la génération MicroPython.
  - `compiler.emj.symbol`
    - `AllSymbol` : représentation d'un identifiant (type, portée, `givenName`, valeur, liste des paramètres).
    - `TableSymbol` : `HashMap` + `Stack` assurant la gestion des symboles et des portées.
- `compiler.exception`
  - Exceptions spécialisées du compilateur, par exemple `ParsingException`, `EMJErrorException` ou `SymbolNotFoundException`, utilisées pour signaler les erreurs lexicales, syntaxiques ou sémantiques détectées durant l'analyse et pour propager proprement les fautes détectées.

Cette organisation modulaire sépare clairement **syntaxe**, **sémantique** et **génération**, ce qui facilite les tests unitaires et la maintenance.

5 Description des classes principales

5.1 EMJVisitor

La classe `EMJVisitor` hérite de `EMJParserBaseVisitor<Object>` et assure l'analyse sémantique de l'AST généré par ANTLR. Elle initialise une instance de `TableSymbol` pour gérer la table des symboles et un `EMJErrorLogger` pour collecter les éventuelles erreurs. Lors de sa visite de chaque nœud, elle vérifie la déclaration préalable des identifiants, contrôle la compatibilité des types (entiers, booléens, tuples, etc.), valide les appels de fonction (correspondance entre signature et arguments) et s'assure que la carte de jeu respecte les contraintes (dimensions, présence d'une et une seule voiture de police, etc.). Les méthodes `enterScope` et `exitScope` sont invoquées pour gérer l'imbrication des portées et prévenir les conflits de noms.

5.2 EMJCodeGenerator

La classe `EMJCodeGenerator`, est le visiteur responsable de la génération du code MicroPython. Elle se base sur un `StringBuilder` interne et maintient un compteur `indentationLevel` pour assurer une indentation correcte du code Python. La variable d'état `lastMove` permet d'optimiser les rotations du robot *cuteBot* en mémorisant la dernière direction exécutée. À chaque visite, ce visiteur convertit les structures de contrôle (`if`, `while`, `for`) en blocs Python et traduit les instructions de déplacement emoji en appels aux méthodes `cuteBot.motors()`, `turnLeft()`, `turnRight()`, etc. Il génère également les définitions de fonctions utilisateur sous la forme de `def` Python et ajoute un appel final à `main()`.

5.3 AllSymbol

La classe `AllSymbol` représente un symbole du programme (variable, fonction ou paramètre). Elle contient les attributs `name` (nom source), `type` (type sémantique), `scope` (portée d'appartenance), `givenName` (nom généré pour le code cible), `value` (valeur littérale ou nulle) et `params` (liste des types de paramètres pour les fonctions). Des méthodes d'accès (`getters/setters`) et `addParams` permettent de construire et interroger ces informations au cours de l'analyse.

5.4 TableSymbol

`TableSymbol` implémente la table des symboles du compilateur à l'aide d'une `HashMap` indexée par le nom d'identifiant et d'une `Stack` gérant les portées lexicales. Les méthodes clés `defSymbol()`, `getSymbolFromID(id, scope)`, `enterScope(scopeName)` et `exitScope()` permettent respectivement d'ajouter un symbole, de le rechercher dans une portée donnée et de gérer l'entrée/sortie de blocs de code. Cette combinaison garantit l'unicité des noms dans chaque portée et fournit les informations nécessaires à la passe de génération.

5.5 Main

La classe `Main` constitue le point d'entrée du compilateur. Elle utilise Apache Commons CLI pour traiter les options `-i` (fichier source EMJ) et `-o` (fichier de sortie Python). Le pipeline d'exécution comprend l'analyse lexicale/syntaxique via ANTLR (avec `MyConsoleErrorListener`), puis la visite sémantique par `EMJVisitor`. En cas d'erreurs, `Main` affiche « KO » sur `stderr` et interrompt le processus. Si tout est valide, il invoque `EMJCodeGenerator` pour produire le code MicroPython et affiche « OK » à la fin.

6 Conclusion

En définitive, la réalisation du compilateur **EMJ** nous a donné l'occasion d'appliquer concrètement les notions de syntaxe et de sémantique des langages de programmation au travers d'un projet tangible. Nous avons mis en place une table des symboles solide afin de gérer efficacement les identifiants et leurs portées, et nous avons motivé plusieurs décisions clés — recours aux structures `HashMap` et `Stack`, répartition claire des responsabilités entre les classes, gestion de l'indentation du code généré, ainsi que le suivi de la dernière action du robot via `lastMove` — pour garantir la fiabilité et la lisibilité de l'outil.

Le projet aboutit à un compilateur opérationnel capable d'analyser du code EMJ et de le traduire en instructions exécutables sur la plateforme visée (le robot associé via MicroPython, ou une simulation interne). Il témoigne de notre compréhension approfondie des concepts théoriques abordés en cours et de notre aptitude à les mobiliser pour concevoir, de façon autonome, un langage spécialisé et son compilateur. Cette expérience de construction de compilateur constitue un atout précieux que nous pourrons réinvestir dans de futurs projets touchant à la création de langages ou au développement d'outils de programmation.