

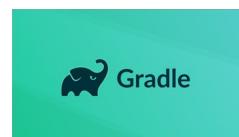
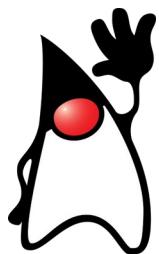
MCDALANG

Un projet de :

Corentin GOUESS
Hugo CASTELL
Arthur LEVI
Alexis WAMSTER
Antoine TEXIER
Ken TIZEN DJONGUE

« Coder n'aura jamais été aussi simple »

Projet Java
1A FISA SN – Promo 2027



Projet Long Java

1A FISA SN

22/06/2025

**Science du Numérique
Informatique et réseaux**
UE Technologie Objet

Examinateur
Xavier CREGUT
Aurélie HURAUT

Sommaire

Titre	1
Sommaire	2
1 Introduction	3
2 Gestion de projet	3
2.1 Méthodologie de gestion de projet	3
2.2 Phases du projet	4
2.3 Organisation de l'équipe	4
2.4 Suivi et pilotage	6
2.4.1 Travail en branches	6
2.4.2 Intégration continue (CI) et qualité	6
3 Présentation technique de l'application	7
3.0.1 Langages de Programmation et Frameworks	7
3.0.2 Gestion de Version	7
3.0.3 Outils de Développement	7
3.1 Architecture	8
3.2 L'interface utilisateur	9
3.2.1 Diagrammes UML	9
3.3 Traduction de langage	10
3.3.1 Outils de reconnaissance de langage (ANTLR)	10
3.3.2 La traduction proprement dite	14
3.4 Les Tests	14
3.4.1 Côté Traduction	14
3.4.2 Côté fonctionnel	15
3.4.3 Côté ANTLR	16
3.5 Difficulté rencontrées et Solutions	16
4 Produit final	17
4.1 Fonctionnalité disponibles	17
4.2 Fonctionnalité non disponibles	17

1 Introduction

Dans le cadre de ce projet de programmation, nous avions la liberté de choisir le sujet que nous souhaitions développer. Notre groupe a opté pour la création d'une application nommée **Mcdalang**.

Mcdalang est une application conçue pour traduire des langages de programmation. Elle permet, par exemple, d'écrire un programme en pseudo-code (langage que nous avons appelé Mcdalang) et de le convertir automatiquement en code Python. Cette approche vise à faciliter l'apprentissage de nouveaux langages de programmation. Un utilisateur maîtrisant déjà Python pourra, par le biais de l'application, visualiser l'équivalent de son code en C, ce qui rend la transition entre les langages plus intuitive.

Dans ce rapport, nous présenterons notre méthodologie de gestion de projet, les choix techniques liés au développement de l'application, une description détaillée du produit final, ainsi qu'un bilan global de notre travail.

2 Gestion de projet

Ce projet, réalisé sur une période de 31 jours a nécessité une certaine organisation afin de le mener à bien et de respecter le jalon final. Notre méthodologie a combiné :

- Une répartition équilibrée des compétences (Code, rédaction du rapport, etc...)
- Un suivi hebdomadaire via Discord
- L'utilisation de Github pour le versionnement

2.1 Méthodologie de gestion de projet

Dès le lancement du projet, nous avons opté pour une approche agile, structurée autour de sprints, afin de garantir une organisation claire et un avancement régulier. Deux premières séances ont été consacrées à cadrer nos objectifs, définir les fonctionnalités attendues et planifier les différentes étapes.

Cette méthode nous a permis dès la première séance de :

- Définir une liste des fonctionnalités attendues et les catégoriser pour cerner les sprints
- Faire des croquis d'UI et débattre du placement des éléments
- Faire des recherches sur les technologies que nous pouvions utiliser

Ainsi, dès la seconde séance, nous avons pu :

- Initialiser le projet
- Réaliser un kanban complet des tâches séparés en sprints
- Initier aux outils ceux qui n'y étaient pas familiers
- Mettre en place les branches de développement

2.2 Phases du projet

Nous avons donc divisé le projet en trois sprints distincts représentant une échelle de temps équivalente à une semaine environ.

- 1 - Application fonctionnelle
- 2 - Ajout de nouvelles fonctionnalités pour compléter l'application
- 3 - Préparation de la présentation orale et derniers ajouts de fonctionnalités

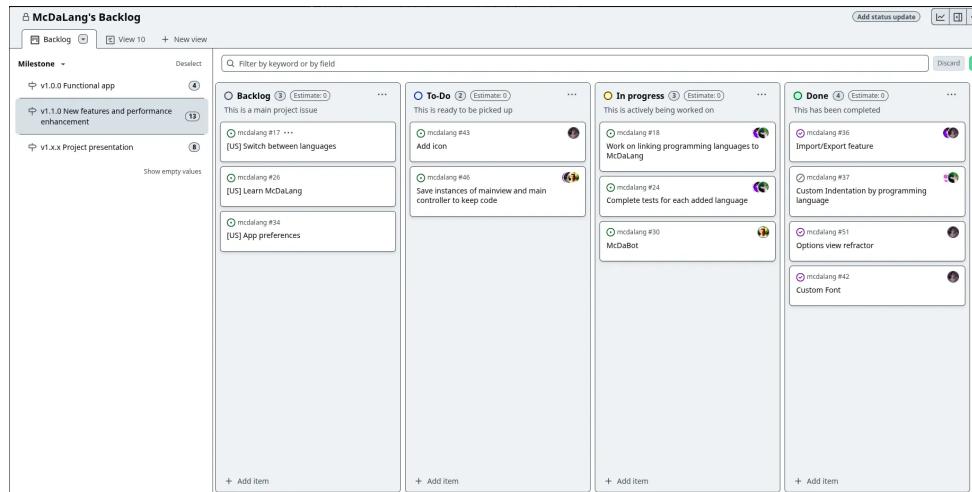


Figure 1. Backlog

2.3 Organisation de l'équipe

Une fois le tableau des tâches prêt, nous nous sommes naturellement tournés vers les tâches qui nous plaissaient le plus et les équipes se sont formées à ce moment-là.

	UI Team	ANTLR Team	Gestion Projet	Test team
Antoine		x		
Arthur		x		x
Alexis	x			
Ken		x		
Corentin	x			
Hugo	x		x	x

Figure 2. Repartition équipe

Les tâches de notre kanban sont classées hiérarchiquement.

- 1 Le plus haut niveau est le Backlog composé de User Stories.
- 2 Les sous-issues de ces User Stories apparaissent dans le kanban dans les colonnes Todo/In progress/Done et nous donne une vision macro de l'avancée du sprint

3 Ces issues macro peuvent elles-même être composées de plusieurs étapes et nous allons ajouter encore un niveau d'abstraction en ajoutant des sous-issues qui viendront donner un indicateur de l'avancée de la tâche.

Format d'une User Story :

- Description : décrit qui, quoi et pourquoi dans la fonctionnalité
- Priorité (importance) : MOSCOW représentation (Must/Should/Could/Would)
- Critères d'acceptation : liste des critères qui définissent que la fonctionnalité est implémenté avec une liste des sous-tâches associés

[US] Switch between languages #17

 Open  1 / 2  Hugo-CASTELL/mcdalang Public

Hugo-CASTELL opened 2 weeks ago · edited by Hugo-CASTELL  Edits  Owner ...

[US] Switch between languages

Status:  Draft

Description

- As an application user
- I want to be able to change languages
- So that I can see how a language translates in another

Importance

Should

Acceptance Criteria & Subtasks

- Grammar links between McDaLang and other languages
 - [Work on linking programming languages to McDaLang #18](#) 
- User documentation can be accessed through the application
 - [\[US\] Basic UI/UX #2](#) 

Please refer to [#11 issue page](#) to acknowledge what to write in the technical documentation



 Sub-issues 

 Work on linking programming languages to McDaLang #18 
 Custom Indentation by programming language #37 

Figure 3. Github

2.4 Suivi et pilotage

Table 1. Outils utilisés dans le projet

Fonction	Outil	Commentaire
Versionnage	Git sur Github	Plus de flexibilité que Gitlab (VPN)
Automatisation de build	Gradle	cf partie CI
Environnement de développement	IntelliJ IDEA	Pour harmoniser
Qualité de code	SonarQube IDE	Vérification de chaque fichier à chaque commit
Tests unitaires	JUnit	cf partie CI
Gestion de projet	Github Projects	

2.4.1 Travail en branches

Le versionnage a suivi un modèle classique :

- **main** : branche stable représentant la version production
- **dev** : branche d'intégration des fonctionnalités validées
- branches de fonctionnalités : créées à partir de **dev**, fusionnées via Pull Requests après revue

2.4.2 Intégration continue (CI) et qualité

La qualité du code et des livrables a été assurée par plusieurs mécanismes :

- **CI avec Gradle** : exécutée à chaque commit, elle lançait automatiquement les tests unitaires.
- **Pull Requests** : chaque ajout de code passait par une revue systématique d'au moins un autre membre.
- **Protection de la branche main** : seuls les commits validés via PR pouvaient y être intégrés.
- **Traçabilité** : chaque commit devait faire référence à l'issue correspondante dans le Kanban.

3 Présentation technique de l'application

Notre stack technique est conçue pour assurer une maintenabilité du code et offrir une expérience utilisateur fluide. Voici les principaux composants de notre stack technique :

3.0.1 Langages de Programmation et Frameworks

- **Java Swing** : Utilisé pour construire l'interface utilisateur de l'application. Couplé avec FlatLaf, cela permet d'avoir des thèmes modulables et faciles à implémenter.
- **GRadle** : Outil de build utilisé pour automatiser la compilation, les tests et le déploiement de l'application. Il nous a également permis d'intégrer facilement des dépendances et de gérer les configurations de build.
- **JUnit** : Framework de test utilisé pour écrire et exécuter des tests unitaires. JUnit aide à garantir que chaque composant de l'application fonctionne correctement et permet de détecter rapidement les régressions.

3.0.2 Gestion de Version

- **GitHub** : Plateforme de gestion de version utilisée pour héberger le code source de l'application. Elle nous permet de faire du versionnage de façon efficace et facilite la collaboration entre les membres de l'équipe.

3.0.3 Outils de Développement

- **SonarQube** : Utilisé pour l'inspection continue de la qualité du code. SonarQube aide à identifier et corriger les bugs, les vulnérabilités et les mauvaises pratiques de codage.
- **ANTLR** (ANother Tool for Language Recognition) : Utilisé pour la reconnaissance et la traduction de langage. ANTLR est un puissant générateur d'analyseurs lexicaux et syntaxiques qui facilite la création de langages spécifiques à un domaine.
- **ANTLR** : Utilisé pour la reconnaissance et la traduction de langage. ANTLR est un puissant générateur d'analyseurs lexicaux et syntaxiques qui facilite la création de langages spécifiques à un domaine.

3.1 Architecture

L'interface utilisateur de notre application est conçue selon le modèle MVC (Modèle-Vue-Contrôleur), avec des vues et des contrôleurs spécifiques pour chaque partie de l'interface.

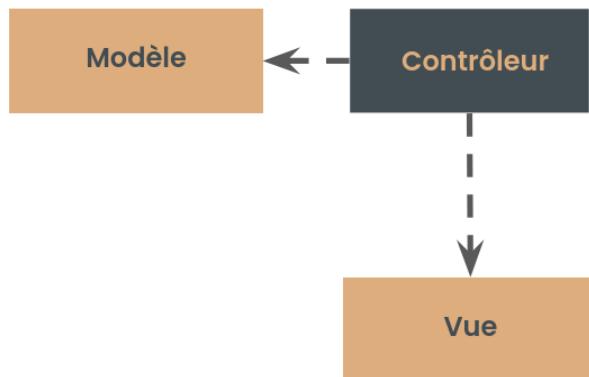


Figure 4. MVC

- **Vue** : Les différentes vues de notre application incluent **MainView**, **SplashView**, **EasterEggView**, **McdaBotMainView**, et **OptionsPopupView**. Chaque vue est responsable de l'affichage d'une partie spécifique de l'interface utilisateur. Par exemple, la **MainView** affiche l'interface principale de l'application, tandis que la **SplashView** affiche un écran de démarrage.
- **Contrôleur** : Les contrôleurs associés à chaque vue gèrent les actions de l'utilisateur et mettent à jour la vue en conséquence. Par exemple, le **MainController** gère les actions de l'utilisateur dans la **MainView**, tandis que le **SplashController** gère les actions de l'utilisateur dans la **SplashView**.
- **Modèle** : Le modèle représente les données et la logique métier de l'application. Il est responsable de la gestion des données et de la logique de traduction des langues. Le modèle utilise ANTLR pour la reconnaissance de langage et la traduction proprement dite **Translate**.
- **Cycle de vie de l'interface utilisateur** : Le cycle de vie de l'interface utilisateur commence par l'initialisation des contrôleurs et des vues. Un contrôleur initialisé peut demander à l'**AppManager** d'afficher sa vue dans la frame principale ou dans un popup. Par exemple, le **MainController** peut demander à l'AppManager d'afficher la MainView dans la frame principale de l'application. À la fermeture de l'application, l'AppManager sauvegarde les préférences de l'utilisateur.

3.2 L'interface utilisateur

3.2.1 Diagrammes UML

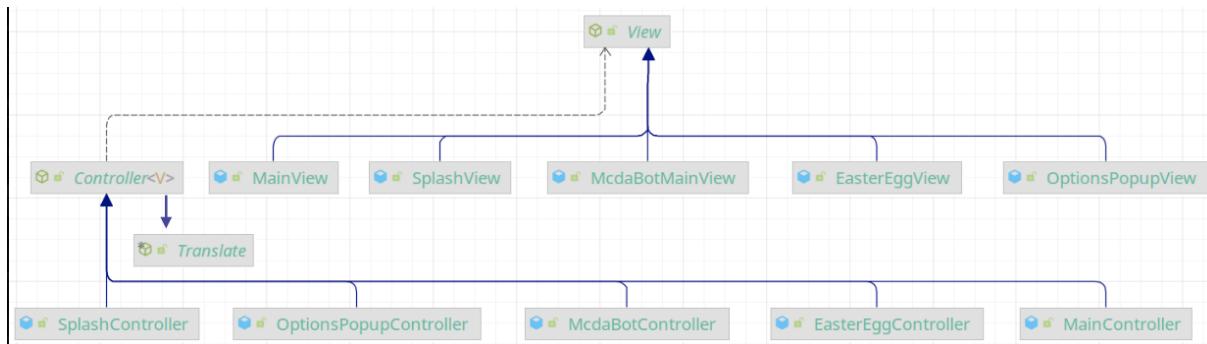


Figure 5. UML

Dans le diagramme UML, on observe une interface **View** qui est implémentée par plusieurs classes. Nous avons différentes vues telles que **MainView**, **SplashView**, **McdaBotMainView**, et **OptionsPopupView**. Comme mentionné précédemment, les vues sont uniquement responsables de l'affichage et ne gèrent en aucun cas les interactions de l'utilisateur.

On peut également voir dans le diagramme que chaque vue a un contrôleur associé. On retrouve une organisation similaire à celle des vues : une interface **Controller** est implémentée par une multitude de contrôleurs. Par exemple, pour la **MainView**, on retrouve le **MainController**.

En ce qui concerne la partie Modèle, le contrôleur interagit directement avec celui-ci. Pour comprendre le fonctionnement du contrôleur, je vous renvoie à la section expliquant le fonctionnement d'ANTLR. La classe 'Translate' sert de passerelle entre le contrôleur et le modèle. Cette classe est responsable de la traduction directe de McdaLang vers un langage cible. Elle instancie le lexer et le parser, utilise l'AST pour effectuer la traduction vers le langage souhaité, et termine par le walker.

3.3 Traduction de langage

Au début de ce projet, le concept de traduction de langage nous était étranger, on le voyait comme quelque chose de réalisable, mais trop ambitieux. Cette vision de l'irréalisable nous a poussés à faire nos recherches sur le sujet, histoire de savoir si de tels projets n'avaient pas déjà été faits et si oui, quel était l'outil central permettant de faire une traduction de langage ou mieux, construire un langage complet en y définissant notre propre syntaxe et nos règles.

C'est donc durant ces recherches que nous avons découvert **ANTRL** (Another Tool for language Recognition), et avons décidé de s'en servir pour créer le pseudo-langage : **Mcdalang**

L'objectif était de se servir de ce pseudo-langage comme langage pivot, et de le traduire dans 10 langages cibles qui sont :

- C
- C++
- Java
- Ada
- Python
- JavaScript
- PowerShell
- Rust
- Ruby
- Assembleur

Nous avons tout de même gardé à l'esprit que le Mcdalang ne devait pas lui-même devenir un langage de programmation, car son but premier était ou est pédagogique.

Dans la suite de cette partie, nous parlerons brièvement de ce qu'est ANTLR, comment il fonctionne et de ce que nous en avons fait pour notre projet.

3.3.1 Outils de reconnaissance de langage (ANTLR)

ANTLR est un constructeur de langage ou un générateur de parseur nécessaire pour valider le fonctionnement d'une grammaire. Il est entièrement écrit en **JAVA**, et possède donc un fichier JAR qui contient tout ce qu'il faut pour compiler et exécuter les reconnaiseurs générés par ANTLR. En résumé, l'outil ANTLR transforme des grammaires en programmes capables de reconnaître des phrases appartenant au langage décrit par la grammaire.

Par exemple, si vous fournissez une grammaire pour JSON, l'outil ANTLR générera un programme capable de reconnaître des entrées au format JSON, en s'appuyant sur certaines classes de la bibliothèque d'exécution ANTLR.

Le fichier JAR contient également deux bibliothèques de support :

- Une bibliothèque avancée pour l'affichage en arbre (tree layout library) ;
- StringTemplate, un moteur de templates utile pour générer du code ou d'autres textes structurés.

Comme une image vaut plus que mille mots, jetons un coup d'œil au pseudo diagramme UML suivant afin de comprendre comment cela marche en pratique.

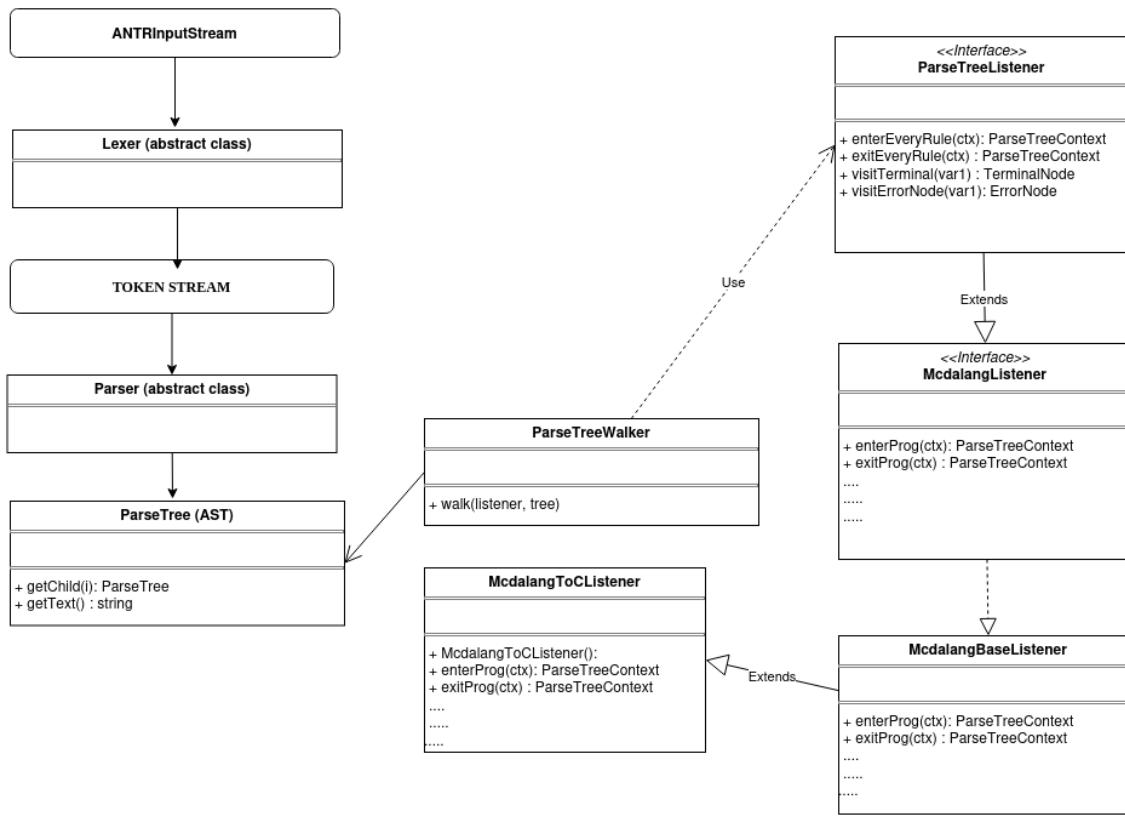


Figure 6. ANTLR fonctionnement

Au début, on récupère le texte source à traduire (ex. un programme dans le langage Mcdalang) et on le stocke dans une variable de type ANTLRInputStream (ou CharStream dans les versions récentes). Ensuite, le traitement se déroule en plusieurs étapes :

- Grâce au **lexer** (analyse lexicale), le texte est transformé en une suite de **tokens** (unités lexicales). Ces tokens sont ensuite transmis au **parser** via un flux de type **TokenStream**.
- Le **parser** (analyseur syntaxique), en s'appuyant sur la grammaire du langage, construit un **arbre syntaxique abstrait** appelé **ParseTree**, qui représente la structure grammaticale du code source.
- Cet arbre syntaxique est ensuite parcouru à l'aide de la classe **ParseTreeWalker**. Cette classe fournit une méthode `walk(listener, tree)` qui prend en paramètre un **listener** (auditeur) et l'arbre à parcourir.
- ANTLR génère automatiquement une classe **BaseListener**, qui implémente l'interface **ParseTreeListener**. Cette interface définit les méthodes de visite (ex. `enterX()`, `exitX()`) que le walker appelle lors du parcours de l'arbre.
- Pour effectuer la traduction, nous créons une classe personnalisée **McDALANGToCListener**, qui hérite de **BaseListener**. Dans cette classe, nous redéfinissons les méthodes nécessaires pour extraire les informations de l'arbre et les convertir en instructions en langage C.

Ci-dessous le début de la grammaire du Mcdalang :

```

1 grammar Mcdalang;
2
3 // Point d'entrée pour le parseur
4 prog
5   : statement+
6   ;
7 // Instructions
8 statement
9   : methodDecl
10  | varDecl NEWLINE
11  | assignment NEWLINE
12  | incrStmt NEWLINE
13  | returnStmt NEWLINE
14  | printStmt NEWLINE
15  | funcCall NEWLINE?
16  | expr NEWLINE
17  | ifStmt
18  | loopStmt
19  | block
20  | NEWLINE
21  ;

```

Au début du fichier, on définit le nom de la grammaire, c'est la directive obligatoire en haut de tout fichier .g4. Ici, la grammaire s'appelle Mcdalang. ANTLR va générer les classes McdalangParser, McdalangLexer, etc.

Ensuite, on définit la règle prog – point d'entrée principal du parseur. Elle indique que le programme complet est composé d'une ou plusieurs instructions (statement+). C'est cette règle qu'on appelle généralement depuis parser.prog() dans le code Java.

Enfin, la règle statement – représente une instruction possible du langage. Chaque alternative correspond à un type d'instruction que le langage peut reconnaître. Voici les cas :

Table 2. Alternatives de la règle **statement** dans la grammaire Mcdalang

Alternative	Signification
methodDecl	Déclaration d'une fonction ou procédure.
varDecl NEWLINE	Déclaration de variable, terminée par un retour à la ligne.
assignment NEWLINE	Affectation d'une valeur à une variable (ex : <code>x = 5</code>), suivie d'un retour à la ligne.
incrStmt NEWLINE	Incrémentation ou modification incrémentale (ex : <code>x++</code> , <code>x += 1</code>).
returnStmt NEWLINE	Instruction de retour d'une valeur depuis une fonction (ex : <code>return x</code>).
printStmt NEWLINE	Instruction d'affichage à l'écran (ex : <code>print("Hello")</code>).
funcCall NEWLINE?	Appel d'une fonction. Le retour à la ligne est optionnel.
expr NEWLINE	Expression isolée sur une ligne (ex : un calcul comme <code>a + b</code>).
ifStmt	Instruction conditionnelle (ex : <code>if, else if, else</code>).
loopStmt	Boucle de contrôle (ex : <code>for, while</code>).
block	Bloc d'instructions regroupées, typiquement entre accolades <code>{ ... }</code> .
NEWLINE	Ligne vide (souvent utilisée pour espacer visuellement les instructions).

On se sert donc de cette grammaire pour écrire le code exemple suivant, et on observe l'arbre généré par le parser, puis on vérifie qu'il a été capable de tout reconnaître grâce aux éléments de grammaire qu'il possède.

```

1 methode entier addition(entier a, entier b) {
2     var entier resultat a + b
3     resultat++
4     afficher(resultat)
5     return resultat
6 }
```

Ci-dessous l'image de l'arbre syntaxique généré par le Parser, ici, on parcourt l'arbre en profondeur rive gauche.

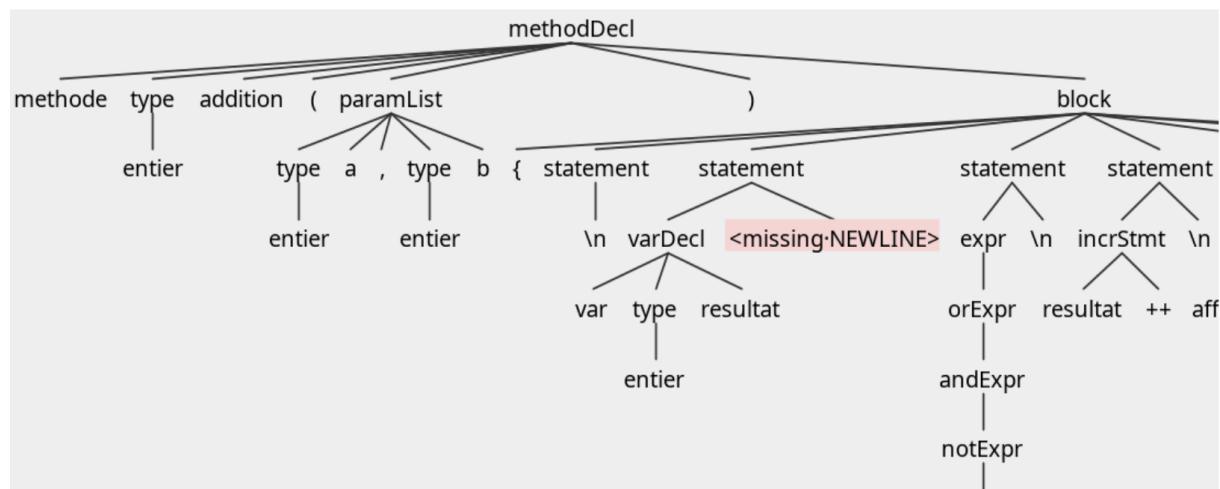


Figure 7. AST généré par le Parser

On voit donc sur cette image que le parser a reconnu la majorité du texte et a remonté une erreur dûe à l'absence d'une nouvelle ligne.

Partant de cet arbre et des méthodes du Listener, on va pouvoir réaliser la traduction proprement dite du Mcdalang vers d'autres langages.

3.3.2 La traduction proprement dite

Comme indiqué précédemment, ANTLR nous permet de générer automatiquement un lexer, un parser, ainsi qu'un listener à partir de la grammaire du langage que nous avons définie.

La traduction repose principalement sur l'utilisation du listener généré par ANTLR, combiné à l'arbre syntaxique généré lorsque le parser d'ANTLR analyse une entrée. Le Parse Tree (arbre de dérivation), qui en résulte reproduit toutes les étapes de l'analyse syntaxique. C'est ce Parse Tree qui est ensuite parcouru par walker d'ANTLR. Celui-ci traverse récursivement l'arbre, et appelle automatiquement, à chaque nœud de l'arbre visité, les méthodes du listener associées à l'entrée (enterX) ou à la sortie (exitX) d'une règle de la grammaire.

Il ne nous reste donc plus qu'à implémenter ces méthodes enterX et exitX dans un fichier java qui hérite de ce listener et adapter la syntaxe que l'on souhaite en sortie.

Les étapes :

- **Génération de l'AST** : On lance d'abord le parser sur un texte en entrée, dans notre cas du code écrit en Mcdalang. Le parser nous ressort l'AST que l'on va utiliser pour faire la traduction.
- **Implémentation du listener personnalisé** : Nous avons ensuite créé une classe héritant du listener généré par ANTLR dans laquelle nous avons redéfini certaines méthodes comme exitVarDecl, exitAssignment, etc. Chaque méthode correspond à une règle de la grammaire Mcdalang, et est appelée lors du passage du walker sur un nœud de l'arbre.
- **Traduction vers le langage cible** : À l'intérieur de ces méthodes, il nous suffit de redéfinir la manière dont on souhaite afficher la traduction, cela consiste globalement à reproduire la syntaxe du langage cible.

3.4 Les Tests

Les tests de l'application ont été réalisés à trois niveaux différents :

- Côté ANTLR afin de s'assurer que le parser reconnaît bien la grammaire
- Côté traduction, car on doit s'assurer que le code traduit en java par exemple, compile bien et respecte les règles syntaxiques du langage.
- Côté UI, les tests étaient plutots fait du point de vu utilisateurs, on s'assurait que toutes les fonctions présentent étaient en état de marche

Dans la suite de cette partie, nous expliciterons un peu plus l'idée derrière ces tests.

3.4.1 Côté Traduction

Afin de garantir la fiabilité du traducteur du langage Mcdalang vers le langage C, une série de tests unitaires automatisés a été mise en place à l'aide du framework JUnit.

L'objectif principal de ces tests est de vérifier que :

- la traduction syntaxique produit un code C conforme et compilable ;
- la sémantique des instructions est respectée : une instruction en Mcdalang doit produire une instruction équivalente en C ;

- les cas limites sont couverts : expressions imbriquées, opérateurs complexes, appels de fonctions, structures conditionnelles multiples, etc.

Chaque test consiste à fournir un code source Mcdalang (sous forme de chaîne ou de fichier) en entrée du traducteur, puis à :

- générer le code C via le listener McdalangToC;
- comparer le résultat obtenu avec une version attendue;
- éventuellement, compiler le code C généré et exécuter un test de comportement (selon le niveau de validation souhaité).

Voici un exemple représentatif d'un test JUnit :

```
1  @Test
2  void declarationTest() {
3      List<Pair<String, String>> declarations = List.of(
4          Pair.of("var entier x = 10", "int x = 10;"),
5          Pair.of("var flottant y = 3.14", "float y = 3.14;"),
6          Pair.of("const chaîne nom = \"Alice\"", "char* nom = \"Alice\";"),
7          Pair.of("var bool estValide = true", "bool estValide = true;"),
8          Pair.of("var char lettre = 'A'", "char lettre = 'A';")
9      );
10
11     for (Pair<String, String> declaration : declarations) {
12         assertEquals(declaration.second.strip(), superTest(declaration.first +
13             "\n").strip());
14     }
}
```

D'autres tests valident :

- les affectations simples ou multiples;
- les opérations arithmétiques (addition, multiplication, puissances);
- les boucles (tantque, pour, faire);
- les conditions (si, sinon, sinon si);
- les fonctions (déclaration avec paramètres et retour, appel de fonction);
- les opérateurs logiques (AND, OR, NOT, opérateurs ternaires);

3.4.2 Côté fonctionnel

À chaque cycle de développement, nous intégrions (merge) l'ensemble des nouvelles fonctionnalités dans une version commune. Pour valider ces ajouts, une réunion était organisée, permettant à tous les membres de manipuler directement l'application.

Cette étape pratique avait pour but de détecter les bugs en conditions réelles d'utilisation. Les participants étaient encouragés à tester librement la nouvelle version, en essayant notamment des cas inhabituels ou inattendus susceptibles de révéler des dysfonctionnements.

Les anomalies identifiées lors de ces sessions étaient consignées sous forme de tickets (issues) dans notre gestionnaire de projet. Une personne dédiée prenait ensuite en charge la résolution de ces bugs, assurant ainsi une amélioration continue de l'application.

3.4.3 Côté ANTLR

Avant de passer à l'implémentation de la traduction, nous avons dû concevoir la grammaire du langage Mcdalang, puis vérifier qu'elle fonctionne correctement. Pour cela, nous avons utilisé grun, un outil fourni avec ANTLR permettant de tester le parser et visualiser l'arbre syntaxique produit (Parse Tree). Qu'est-ce que grun ?

grun est un outil en ligne de commande qui permet :

- d'exécuter le lexer et le parser générés à partir d'une grammaire ANTLR ;
- de tester un programme d'exemple écrit dans le langage défini (ici Mcdalang) ;
- d'afficher graphiquement le Parse Tree pour une règle de départ donnée (dans notre cas, la règle **prog**).

Étapes d'utilisation de grun :

Voici les étapes que nous avons suivies pour tester la grammaire à l'aide de grun :

Génération des fichiers Java à partir de la grammaire :

```
1 antlr4 Mcdalang.g4
```

Cela génère automatiquement les fichiers nécessaires : McdalangLexer.java, McdalangParser.java et d'autres fichiers annexes.

Compilation des fichiers générés :

```
1 javac Mcdalang*.java
```

Exécution de grun pour visualiser l'arbre syntaxique :

```
1 grun Mcdalang prog test.mcda -gui
```

- Mcdalang : nom de la grammaire (sans extension .g4)
- prog : règle de départ (la racine de l'analyse)
- test.mcda : fichier d'entrée contenant un code à analyser
- -gui : option permettant d'afficher le Parse Tree via une interface graphique

Une fois cette commande exécutée, une fenêtre graphique s'ouvre avec l'arbre généré par le parser selon les règles définies dans la grammaire. Cela nous a permis de vérifier progressivement le bon fonctionnement de chaque règle en visualisant les branches de l'arbre généré.

3.5 Difficulté rencontrées et Solutions

- Nous avons eu des problèmes avec le caractères de fin de lignes ; de manière générale, les langages de programmations utilisent le point-virgule, sauf que dans notre cas, ce point-virgule peut aussi être utilisé dans la boucle **for**. Nous avons donc décidé d'utiliser le **EOF** comme caractères de fin de ligne afin de s'affranchir d'un traitement particulier pour les boucles **for**
- L'indentation à aussi été un problème, en effet les langages de programmation n'ont pas tous le même niveau d'indentation, on devait donc s'assurer que chaque langage reçoive l'indentation qui lui est propre. Dans un premier temps, nous avons penser à créer une classe abstraite qui posséderait des méthodes abstraite nécessaire à l'indentation; puis, finalement, nous avons opter pour un traitement particulier directement dans les fichiers de traductions des langages concernés.

- La dernière chose est la prise en main d'ANTLR, en effet nous nous sommes servi du livre de son développeur qui faisait à peu près 350 pages. Cette partie a donc été fastidieuse au vu des connaissances qu'il fallait emmagasiner afin de mener à bien ce projet.

4 Produit final

4.1 Fonctionnalité disponibles

Table 3. Fonctionnalités disponibles dans Mcdalang

Catégorie	Fonctionnalités
Langages supportés	Mcdalang, Python, C, C++, Ada, Assembleur, Java, JavaScript, PowerShell, Ruby, Rust
Reconnaissance syntaxique	Déclaration de variables/constantes, boucles (tant que, faire tant que, pour), conditions (si, sinon, sinonsi), fonctions, égalités (égal, différent), comparateurs (>, >=, <, <=), opérations (addition, soustraction, multiplication, division, division entière, modulo, puissance), incrémentation/décrémentation, affichage (print), concaténation de chaînes
Options utilisateur	Changement de taille du texte, mode auto-run (traduction automatique), importation/exportation de fichiers
Support pédagogique	McdaBot : bot d'aide à l'apprentissage du Mcdalang, explication des notions, exemple de programme complet

4.2 Fonctionnalité non disponibles

Table 4. Fonctionnalités non disponibles dans Mcdalang

Fonctionnalité	Commentaire
Surlignage syntaxique	Permettrait un affichage coloré et plus lisible du code
Traduction multilangage	Actuellement, la traduction ne peut se faire que depuis Mcdalang ; nous aurions voulu traduire directement entre deux langages existants (ex : Python vers C)
Programmation orientée objet	Fonctionnalité absente, empêche la gestion de concepts avancés comme les classes et objets
Autres notions de programmation	Notamment la gestion des tableaux et d'autres structures de données avancées