ECOLE SUPERIEURE
D'INGENIEURS DE RENNES

Université de Rennes

Rapport de projet

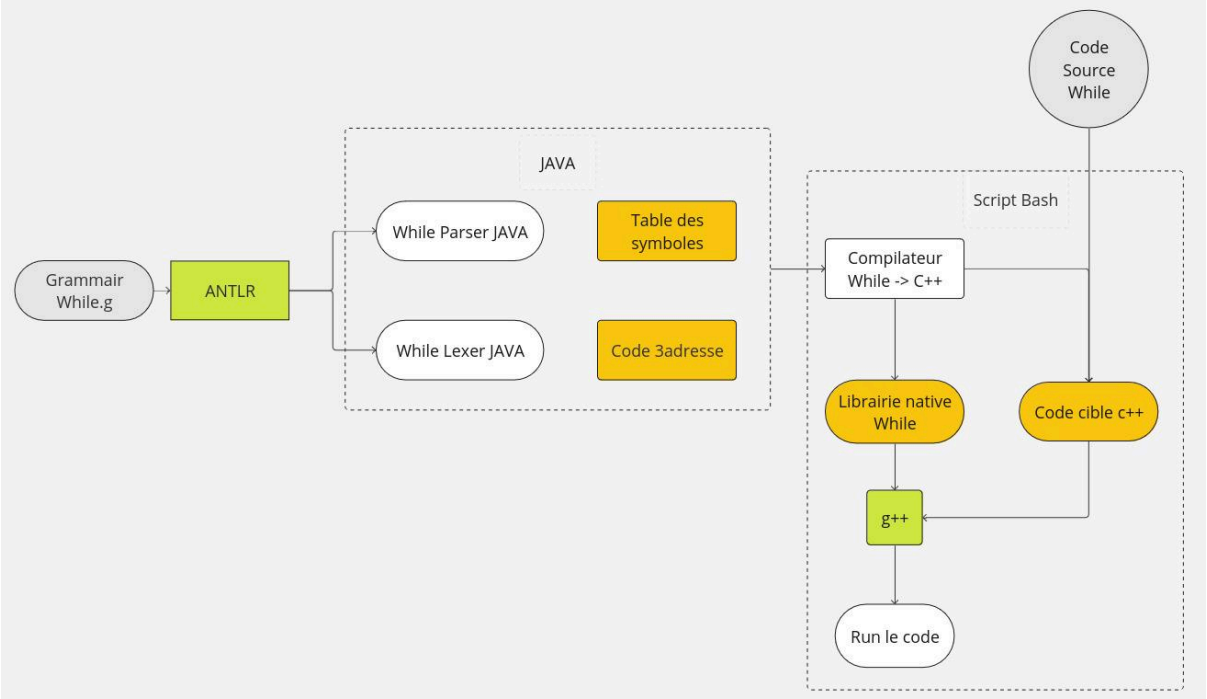
Auteurs :

Noam GEFFROY
Pol JAOUEN
Flavien LEBORGNE
Stevan METAYER
Fanny SHEHABI

2023 - 2024

Description technique

La conception de ce projet à nécessité l'utilisation de plusieurs langage et logiciel afin de suivre le schéma suivant :



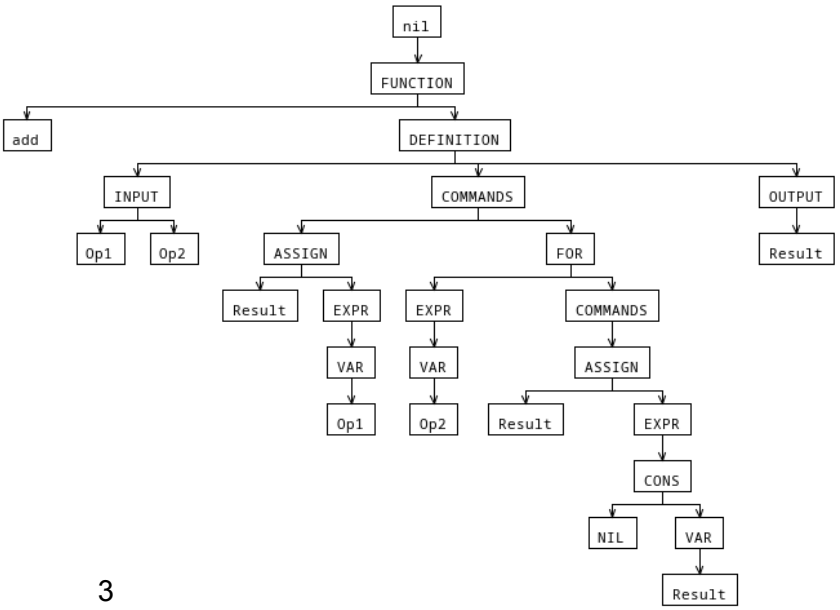
Étape 1 : grammaire While.g

La première étape consiste en la réécriture de la grammaire donnée en Antlr, pour la génération de l'AST. Nous avons choisi d'ajouter différents tokens (FUNCTION, ASSIGN, VAR…), afin d'avoir des étiquettes dans notre AST, facilitant par la suite la lecture et l'analyse. L'ajout de @rulecatch, @parser::members, @lexer::members nous permet de gérer nous même les remontées d'erreur du lexer et du parser Antlr, et de les afficher par la suite de manière plus lisible.

Étape 2 : AST sur ANTLR

Pour le code suivant on obtient l'AST

```
function add:
read Op1, Op2
%
Result := Op1;
for Op2 do
Result := (cons nil Result)
od
%
write Result
```



L’arbre commence par un bloc “nil” qui peut avoir un nombre indéfini d’enfants car un programme while peut contenir plusieurs fonctions.

On passe au bloc “**FUNCTION**” qui possède 2 enfants, son nom et un bloc “**DEFINITION**” qui à lui même 3 enfants. Un bloc “**INPUT**” qui contient la liste des variable d’entrée, un bloc “**OUTPUT**” qui contient la liste des variable de sortie et un bloc “**COMMANDS**”.

Le bloc “**COMMANDS**” possède lui aussi un nombre indéfinis d’enfants car c’est la liste des instructions du bloc dans lequel on se trouve. Prenons l’exemple de notre cas, le 1er bloc de commandes contient les blocs “**ASSIGN**” et “**FOR**” alors que le 2e présent dans la boucle for ne contient que “**ASSIGN**”.

Les blocs se trouvant dans commande sont binaire tel que le “**ASSIGN**” qui attribue une expression as une variable, ou le “**FOR**” qui utilise une expression comme condition et exécute un bloc “**COMMANDS**” a chaque itération. Cette formule est plus ou moins la même pour tous les autres blocs de commandes tel que la boucle while et foreach.

Le bloc de condition “**IF**” est un peu différent car il possède 2 ou 3 enfants, cette structure avec un enfant optionnel gère si il y a un else ou non.

Il reste les blocs unaires tel que “**EXPR**”, “**VAR**” qui servent à étiqueter les variable à utiliser mais prennent aussi en compte le découpage des arbre écrit directement dans le code avec les blocs “**CONS**” et “**NIL**”.

Tout cela permet à ANTLRwork de générer un Parser et un Lexer reprenant toutes ces informations en java.

Étape 3 : Table des symbole et code 3 adresses

Le langage while possède plusieur caractéristiques permettant de simplifier la représentation de la table des symboles:

- Chaque variable est locale à sa fonction
- On ne peut pas imbriquer de fonctions
- Une fonction ne peut pas être définie deux fois

Nous avons donc choisis de représenter chaque fonction par un objet contenant son nom, ses entrées et sorties, et les variables qui sont utilisées au cours de l'exécution de cette fonction. La table des symboles est alors une liste contenant les représentations de toutes les fonctions, que l’on construit au fur et à mesure que l’on parcours l’AST. On utilise ensuite cette table des symboles pour s’assurer que l’on ne déclare pas deux fonctions avec le même nom, et pour la génération de code, que toutes les variables déclarées à un moment dans la fonction sont bien initialisées.

Code 3 adresses :

Le langage intermédiaire utilisé dans ce projet est un code 3adresse que l'on peut résumer comme une liste de quadruplets. Un quadruplet est un objet avec 4 chaines de caractère, un nom, un résultat et 2 arguments.

Le résultat prend la forme de la chaine "op[i]" avec i un nombre qui s'incrémente à chaque nouveau quadruplet. Les 2 arguments sont utilisés dans le cas des bloc opérateur binaire / unaire afin de pointer les résultats des objets ciblés.

Cela donne une table de la forme suivante : *NOM : arg1, arg2 -> résultat*

```
FUNC_BEGIN: main, null -> op0
VARIABLE: Op1, null -> op1
INPUT: Op1, null -> op2
VARIABLE: Op2, null -> op3
INPUT: Op2, null -> op4
VARIABLE: Result, null -> op5
VARIABLE: Op1, null -> op6
EXPR: Op1, null -> op7
ASSIGN: Result, Op1 -> op8
COMMAND: op8, null -> op9
VARIABLE: Op2, null -> op10
EXPR: Op2, null -> op11
FOR: Op2, null -> op12
VARIABLE: Result, null -> op13
NIL: null, null -> op14
VARIABLE: Result, null -> op15
CONS: op14, Result -> op16
EXPR: op16, null -> op17
ASSIGN: Result, op17 -> op18
COMMAND: op18, null -> op19
END_FOR: null, null -> op20
COMMAND: op20, null -> op21
VARIABLE: Result, null -> op22
STRING: int, null -> op23
VARIABLE: Result, null -> op24
CONS: op23, Result -> op25
EXPR: op25, null -> op26
ASSIGN: Result, op26 -> op27
COMMAND: op27, null -> op28
VARIABLE: Result, null -> op29
OUTPUT: Result, null -> op30
FUNC_END: null, null -> op31
```

On peut donc y voir **2 types de lignes**, les **lignes de structure** comme la ligne 1 servant à découper les blocs fonctionnels du code et les **lignes d'information** comme la ligne 2 transmettant le nom de la variable ou les commandes utilisées. Cela permet de passer à l'étape de l'écriture du code.

Étape 4 : Génération de code à partir du code 3 adresse

Nous avons choisi comme langage cible le c++. Pour générer le code, nous parcourons le code 3 adresse instruction par instruction. Nous avons une classe abstraite "Instruction" dont vont hériter des classes représentant chaque instruction. La méthode toString de chacune de ces classes réalise la conversion du quadruplet du code 3 adresse en chaîne de caractère en ligne de code c++. Une classe "Bloc" hérite d'"Instruction", et sert de classe mère pour toutes les instructions représentant un bloc (fonction, boucles for et while, if). Tous les blocs ont la capacité de contenir une suite d'Instruction, et ont aussi une méthode toString, permettant d'obtenir leur représentation en c++. Lors du parcours du code 3 adresse, on stocke dans une liste les objets "Fonction", dans une pile les objets "Bloc", et

on ajoute au dernier bloc de cette pile les instructions que l'on rencontre. Lorsque l'on rencontre une instruction de type "END_IF" ou "END_FOR", on retire le dernier bloc de la pile.
À la fin du parcours du code 3 adresse, on vient écrire dans un fichier les différent "#include", puis on y ajoute la conversion en String de chaque objet "Fonction", qui génère tout le code. La fonction nommée "main" est générée de manière légèrement différente, de manière à suivre la mise en forme d'une méthode "main" c++, permettant d'avoir un point d'entrée au programme.

Bibliothèque runtime

La bibliothèque runtime est écrite en c++, et consiste en une unique classe "Node", avec les deux fichiers "Node.h" et "Node.cpp". Les Nodes représentent la structure d'arbre binaire du langage while, et permettent les différentes opérations nécessaires au langage. Chaque Node contient une chaîne de caractère représentant son éventuel symbole, ainsi que deux shared pointers vers ses fils gauche et droit. Une node est une feuille si les deux shared pointers sont des nullptr. Nous avons implémenté les méthodes asBoolean (false si la Node est une feuille, false sinon), asInteger (Taille de l'arbre en ne parcourant que les fils droits en partant de la racine), et asString (Concaténation des symboles des feuilles de l'arbre), permettant les conversions des arbres en types c++ utilisables pour les comparaisons, les boucles et l'affichage. De plus, deux méthodes fromInt et fromString permettent de construire un arbre à partir d'un entier et d'une chaîne de caractère, pour les entrées utilisateur. Enfin, la méthode pp est l'implémentation du pretty printer, qui permet l'affichage dans la sortie standard de différentes représentations des arbres (entier, booléen, chaîne de caractère).

Étape 5 : Maven + Scripts Bash

Afin de rendre ce projet fonctionnel, la partie compilation en java doit produire un exécutable en .jar. Pour cela nous utilisons Maven et son plugin "assembly". Un premier script Bash est donc utilisé pour simplifier la commande "mvn clean install assembly:single".

Un deuxième script est défini dans la librairie c++ pour automatiser l'utilisation de g++ et construire le programme. Il nécessite le compilateur.jar, les fichiers de la bibliothèque ainsi que l'adresse du fichier à compiler.

Finalement 2 scripts principaux sont à la racine du projet. Le script "compilateur.sh" et le script "compilateur_sans_echec.sh". Le plus simple des 2 étant celui sans erreur car il contient le lancement du script Maven et le déplacement du compilateur.jar dans la librairie puis l'exécution du script de la librairie. Mais il est recommandé d'utiliser le script standard dans le cas ou le compilateur n'as pas été altérée car il va vérifier l'existence du compilateur.jar et ne le construire que si manquant. Cependant si le code java à été altéré les modification ne seront pas prisent en compte si le compilateur à déjà été compilé avant.
Il vérifie aussi l'existence du fichier source while.

Le programme while sera exécuté lors de la compilation mais un fichier exécutable sera aussi généré à la racine du projet sous le nom “programme”.

Un dernier script “run_test_coverage.sh” a été créé afin de pouvoir lancer les tests et vérifier leurs couvertures. Une fois le script lancé, vous pouvez vous rendre dans le répertoire “target” puis “site” et enfin ouvrir le fichier “index.html”. Une fois le site ouvert, rendez vous sur Jacoco puis suivez les différents liens jusqu'à trouver la couverture.

Description de la validation du compilateur

Parmi les fonctionnalités non implémentées, on peut noter l’assignation de plusieurs variables et la gestion des fonctions à plusieurs variables de retour.

Nous avons effectué une légère optimisation du code 3 adresses (passage de la variable contenant la condition des IF, FOR ou WHILE dans l’argument de l’instruction du code 3 adresse), cela aurait pu être davantage développé.

Afin de s’assurer du bon fonctionnement du compilateur à chaque étape, nous avons créé deux suites de tests. Le premier se chargera de tester la grammaire tandis que le second se chargera de tester les méthodes Java du compilateur.

La première suite de test a donc pour objectif de s’assurer que le programme entré par l’utilisateur en while soit correct dans le sens syntaxique et lexical. Ces tests se composent de la façon suivante, on écrit un programme en correct sur lequel on va appeler le lexer et le parser. Ainsi, si aucune exception n’est levée c’est que le parser et le lexer comprennent et acceptent le langage while. A contrario, on fait la même chose avec un code while qui ne respecte pas les spécifications et on attend du lexer ou du parser de renvoyer une exception.

La deuxième suite de test se concentre sur le code Java, une suite de tests simples afin de vérifier que les fonctions créées renvoie bien ce que l’on attend d’eux comme pour la création d’une stack par exemple.

Concernant la validation du compilateur après compilation, les tests sont fait en passant à la main des paramètres au programme et à vérifier que le résultat attendu corresponde avec le résultat donné après exécution de ce dernier. Par exemple, on compile une fonction while qui additionne deux nombre, puis on vérifie que l’addition est correcte pour plusieurs paramètres différents.

Afin de pouvoir vérifier la couverture de nos tests unitaires, nous avons le plugin Jacoco au Maven. Jacoco permet de visualiser clairement et de donner un pourcentage de la couverture des tests. (se référer à la partie “maven+script bash” pour lancer Jacoco).

Ainsi, nos tests ne couvrent pas l’entièreté de notre application, cependant, ils sont suffisants pour donner l’assurance que notre compilateur fonctionne dans le cadre de la compilation du langage while.

Description de la méthodologie de gestion de projet

Afin d'assurer une bonne collaboration entre les membres, on a utilisé Git. Cela permet à la fois de faire différentes versions incrémentales du projet ainsi que de travailler à plusieurs sur différentes tâches grâce à Github qui permet de partager un projet Git.

Pour assurer une bonne communication entre les membres du groupe, nous avons utilisé Discord qui permet d'envoyer des messages et d'échanger lorsque quelqu'un rencontre un problème ou veut poser une question.

On a réparti les tâches en fonction des fonctionnalités que devait avoir le compilateur. On a choisis de les réaliser dans le même ordre que le compilateur l'exécute afin d'avoir un état d'avancement du projet. On s'est aussi assuré de ne pas laisser quelqu'un tout seul sur une partie afin que tout le monde puisse échanger lorsqu'il rencontre un problème sans avoir à tout réexpliquer à chaque fois.

Rapport Individuel Pol Jaouen

Avec Fanny, je me suis tout d'abord occupé de la création d'une première version de la grammaire. Cette version reconnaissait les instructions de base, et construisait un AST. Nous avons aussi choisi d'ajouter des tokens supplémentaires, pour rendre l'AST plus lisible et plus facile à parcourir pour la suite du projet.

Ensuite, après avoir bataillé pour faire fonctionner les classes java générées par AntlrWorks, j'ai réalisé l'implémentation de l'Analyseur Sémantique. Nous avons longtemps hésité sur l'implémentation, notamment avec l'utilisation de Spaghetti Stack. Mais après réflexion, nous nous sommes rendu compte que le langage while ne nécessitait pas de structure de données aussi complexe, et nous avons utilisé notre propre implémentation.

En attendant les premières versions fonctionnelles du code 3 adresse, je me suis occupé de créer les premières versions de la librairie runtime, avec les opérations de base (ajouter un nœud, supprimer un nœud, convertir un arbre en entier/booléen/chaîne de caractère). J'ai choisi d'utiliser des shared pointers pour simplifier la gestion de la mémoire. En parallèle, j'ai écrit un fichier de test en c++ me permettant de vérifier le bon fonctionnement de mon implémentation des arbres binaires. Quand une première version du code 3 adresses fut finie, je me suis mis à réaliser la génération de code, avec la création des différentes classes pour chaque instruction, et l'affichage dans la console du code généré. J'ai implémenté les différentes instructions de manière incrémentale, en commençant par celles des fonctions *true* et *false*, et en ajoutant au petit à petit des instructions plus complexes (boucles, conditions…). J'en profitais pour réaliser quelques optimisations au code trois adresses, comme le passage de la variable contenant la condition ou le nombre d'itération en paramètre des instructions IF, FOR et WHILE. Après avoir obtenu une première version de code généré fonctionnelle, j'ai laissé la main à mes coéquipiers pour les dernières fonctionnalités qui permettent au compilateur d'enfin prendre forme et d'être utilisable.

Ce projet m'a permis de mieux comprendre le fonctionnement des compilateurs et des grammaires Antlr. Il m'a aussi permis de reprendre le c++, langage que je n'avais pas eu l'occasion de pratiquer depuis les cours de l'année dernière.

Rapport Individuel Noam Geffroy

Après avoir compris le langage While et tester la grammaire, j'ai rejoint l'équipe de l'AST afin de produire le plus vite possible un Parser et un Lexer en java.

Dans un 2e temps j'ai configuré tout le projet Maven avec Fanny pour implémenter les tests et le checkstyle.

Une fois cette étape passée j'ai été dédié à la création du code 3 adresse. La forme que prendra cette liste à très vite été choisie mais de nombreuses séances ont été requises afin de pleinement comprendre à quoi devait ressembler ce langage intermédiaire. Une fois le concept compris je suis passé à la 1ere implémentation qui comprenait diverses optimisation avec des aller retours dans la liste. Cependant, optimiser ce code ne le rendait pas plus simple à traduire et donc j'ai écouté les demandes de l'équipe écriture. Cette collaboration aboutit en la version actuelle du code 3 adresses, une liste bien plus longue avec des lignes permettant de construire la structure et d'autres permettant de transmettre des informations concrètes tel que des nom de variable ou les méthodes utilisées.

Le code 3 adresses a subi divers ajustement et amélioration en parallèle de l'avancement du projet pour l'équipe qui utilisait le code intermédiaire pour écrire le code cible. Suite à ça je suis passé à l'utilisation des paramètres d'entrée. J'ai donc modifier la partie écriture de fonction pour créer le cas spécial de la méthode "main". J'ai implémenter les méthodes qui permettent de traduire les entrées (int ou string) en Node puis de créer les premiers objets de la méthode "main" et de vérifier leur nombre.

Une fois cela fonctionnelle j'ai implémenter l'utilisation d'un fichier texte externe pour stocker le code source while. Et pour finir j'ai paramétré le pom.xml de maven afin de compiler la partie du projet en java.

A cette étape nous avons un programme en java capable de convertir un fichier txt qui contient du code while en un fichier .cpp et une librairie qui contient tous les outils dans langage cible. Il me restait donc à produire les 3 scripts qui lient les différentes parties du projet. et qui servent de premier exécutable à la racine du projet.

Pour finir j'ai participé à l'implémentation de la couverture des tests Jacoco sur Maven, j'ai implémenter les tests liés au code 3adresses puis j'ai rédigé les fichiers d'exemples en While dans le dossier Exemples.

Pour conclure, ce projet m'a permis de comprendre l'intérêt d'un code intermédiaire personnel, lui donner une forme qui ne prend pas en compte le langage cible et donc le rendre accessible à plusieurs langages.

Rapport Individuel Stevan Metayer

J'ai commencé par créer le projet git et ajouter la grammaire While.g dans ce dernier. J'ai ensuite créé le projet java utilisant le Lexer et le Parser servant à générer le code 3 adresses. J'ai pu ensuite écrire des tests afin d'assurer le bon comportement des différents composants utilisés dans le projet java.

La grammaire ne détectant pas les espaces et renvoyant un warning, j'ai essayé de faire en sorte que la grammaire les lisent sans les prendre en compte. Cependant en voulant intégrer la nouvelle grammaire au projet, le programme tournait à l'infini dans le vide sans savoir pourquoi. J'ai donc abandonné tous mes changements. J'ai ensuite aidé à différents

endroits dans la While-Lib (projet c++) lorsque quelqu'un avait besoin (pour les pointeurs par exemple ou le pretty printer).

Nous avons mis en place une suite de test sur le projet Java à l'aide Jacoco qui s'intègre avec Maven. Maven sert aussi à gérer les dépendances (notamment antlr). Il est possible de voir le pourcentage de couverture qu'ont les tests avec le site généré par maven avec les données de test récupéré par Jacoco.

Rapport individuel Flavien Leborgne

Mon projet a commencé avec la découverte du langage while qui était un peu dur à prendre en main, notamment la compréhension des types fait uniquement d'arbre binaire.

Une fois avoir bien compris le langage, j'ai créé différents programmes en while afin de s'en servir comme test et pouvoir vérifier que la grammaire était correctement écrite dans antlrWorks puis je les ai implémenter dans une suite de test Junit. Puis nous avons ajouté Jacoco pour la couverture de test (difficultés à le lancer). J'ai ensuite comme beaucoup de personnes dans le projet à ce moment, essayer d'implémenter le lexer et le parser sur Java, chose que nous avons fini par réussir.

Il a ensuite fallu penser et implémenter la table des symboles. Cette table des symboles nous a permis de réaliser l'analyse sémantique pour laquelle il a fallu définir les règles.

Ma dernière participation au projet hormis la rédaction du rapport fut de créer le pretty printer permettant un affichage plus propre et compréhensif des variables while (un chiffre vaut mieux qu'un arbre binaire).

Rapport individuel Fanny Shehabi

Dans un premier temps, Pol et moi avons créé une première version de la grammaire. Dans la suite logique de cela, nous avons ajouté des lexèmes imaginaires afin de générer un AST le plus lisible et facile à parcourir.

Dans un second temps, j'ai collaboré avec Noam pour la mise en place du projet Maven et du Checkstyle. Nous avons rencontré quelques difficultés.

Ensuite, j'ai pris part aux discussions sur l'élaboration de l'analyseur sémantique avec Pol et Noam. Nous avons longuement hésité sur l'implémentation. Initialement, nous souhaitions utiliser une Spaghetti Stack mais nous nous sommes rendus compte que ce n'était pas nécessaire. Nous avons donc choisi de faire "à notre sauce". Pol s'est occupé de l'implémentation, après ces discussions.

Après cela, j'ai contribué au code 3 adresses en pair programming avec Noam pendant de nombreuses heures. Celui-ci a, au cours du projet, beaucoup évolué et pris plusieurs formes.

Finalement, je me suis occupée de divers morceaux des livrables notamment la documentation utilisateur du langage (la partie *Comment écrire un programme en langage While ?*)

Post Mortem

L’organisation du projet à été plutôt bien maitrisée, la répartition des tâches, l’avancement du projet et la communication entre les membres s'est bien déroulé. Des outils collaboratifs tels que discord, google doc, Github ont permis de suivre et d’organiser au fur et à mesure le projet.

Les plus grosses difficultées rencontrées ont été la compréhension du langage while et la conception du code 3 adresse. C’est 2 parties sont les principales nouveautés de ce projet ce qui explique leur difficulté. Avec le recul et l'expérience, le temps passé sur ces tâches serait bien plus court.