CR - Projet de TLC

LE DOUARIN Marius
BOUGER Yanis
DE ZORDO Benjamin
[INFO - SI]

Table des matières

Introduction

L'objectif de ce projet de théorie des langages et compilation est de pouvoir concevoir un compilateur pour un langage défini en suivant toutes les étapes et pratiques vues en cours lors du premier semestre d'ESIR 2.

Dans ce cadre nous avons dû analyser un document représentant les spécificités du langage While dont nous devions concevoir le compilateur. Celui-ci contenait la grammaire du langage ainsi qu'une explication de son fonctionnement.

While est un langage qui manipule uniquement des arbres binaires dont les feuilles (dernière partie de l'arbre) sont représentées par un lexème Symbol. D'autres expressions nous permettent de décrire ces arbres et certaines commandes nous permettent de les manipuler.

I. Définition de la grammaire sur ANTLR

a. ANTLR : qu'est-ce ?

ANTLR pour *« ANother Tool for Language Recognition »* est un outil d'analyse qui permet la reconnaissance d'une grammaire et de l'interpréter. Il définit un méta langage utilisé pour écrire le fichier grammaire (while.g).

Dans un premier temps, on lui fournit une grammaire logique et structurée : on définit le langage. Ensuite on lui associe une entrée qui est un script dans le langage décrit, et ANTLR nous renvoie l'output défini par la grammaire.

b. Description de la grammaire

Pour la description de la grammaire, nous avons d'abord réécrit les lexèmes et les non terminaux comme indiqué dans le dossier de spécification du projet.

Les lexèmes sont des formules permettant la reconnaissance de certains patterns. Par exemple en supposant que :

- *MIN* est l'ensemble des lettres minuscules (a, b, c … z),
- *MAJ* est l'ensemble des lettres majuscules (A, B, C … Z),
- *DEC* est l'ensemble des chiffres (1, 2 … 9).

Nous pouvons définir le lexème représentant la « variable » de la grammaire While grâce à une expression régulière :

```
VARIABLE
    :  MAJ(MAJ|MIN|DEC)*('!'|'?')?
    ;
```

Les lexèmes MIN, MAJ et DEC sont spéciaux, ils sont appelés fragments et peuvent être utilisés pour définir d'autres lexèmes.

S'ensuit un temps d'analyse afin de bien comprendre à quoi font référence chaque partie de la grammaire, en nous aidant notamment des exemples que nous avions en fin de sujet.

c. Ajout de tokens et construction de l'AST

À cette étape, nous obtenons un arbre de dérivation syntaxique lorsque nous analysons un script en while. Pour pouvoir utiliser le résultat de l'analyse syntaxique plus tard, il a fallu que nous traduisions l'arbre de dérivation syntaxique en un arbre de syntaxe abstrait (AST) ne contenant que les informations nécessaires.

Nous avons pour cela créé un ensemble de tokens, défini en haut de la grammaire ANTLR par une balise « tokens{} ». Les tokens nous permettent de décrire à ANTLR ce qu'il doit afficher dans l'arbre de syntaxe abstrait et quel nom donner aux nœuds.

Si nous prenons par exemple le non terminal « function » :

```
function
    :   'function' SYMBOL ':' definition->^(FUNCTION ^(FUNC_NAME SYMBOL) definition)
    ;
```

nous obtenons cet AST et retrouvons les dénominations précisées par nos tokens.

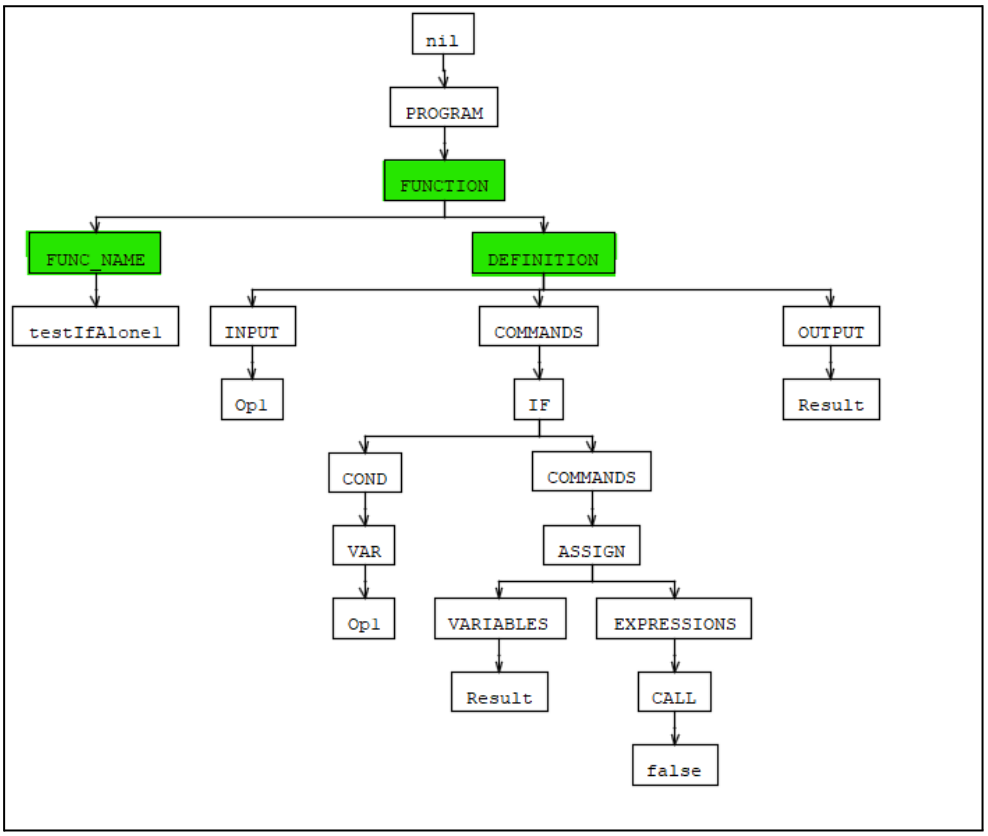

Figure 1 : Arbre de syntaxe abstraite pour un l'exemple de fonction

Ces tokens nous permettent d'organiser notre AST afin de rendre la tâche d'analyse sémantique plus simple notamment pour la création de la table des symboles, mais aussi de faciliter la génération du code intermédiaire (code trois adresses).

II. Analyse sémantique du programme

L'analyse sémantique permet de s'assurer que le programme ait un sens. Elle permet d'invalider le plus possible de programmes étant corrects d'un point de vue syntaxique, mais quand même mauvais (par exemple des programmes n'ayant pas de fonction « main »). Pour ce faire, nous parcourons l'AST en analysant ses nœuds.

La création d'une table des symboles est un processus important de la phase de compilation puisqu'elle a une fonction accélératrice, généralement créée lors de la phase d'analyse lexicale et syntaxique du processus de compilation, elle fait l'intermédiaire. Le principe est de stocker des informations sur les symboles d'un programme, comme ses variables, ses fonctions et ses constantes. Au fur et à mesure de son parcours, le compilateur rencontre des symboles dans le code source, et ces données vont être stockées dans une map qui permet de rechercher rapidement un symbole en fonction de son nom dans une fonction.

Ainsi dans notre main *App.java* nous lisons le contenu du fichier donné en paramètre grâce à un *StringStream* d'ANTLR, *StringStream* que nous passons dans le lexer du langage *while*, à partir duquel nous allons pouvoir en récupérer les tokens. À partir de ces tokens nous pouvons construire le parser du langage *while* en récupérant ce qui est retourné par la méthode *program* du parser. Une fois le parser du *while* bien défini, nous pouvons en récupérer l'arbre résultant, il s'agit de l'AST comme nous pouvons l'observer dans ANTLRWorks, mais où il est désormai possible de le parcourir en largeur dans un premier temps avec le fichier Visitor.java, et un profondeur dans une étape future.

La structure de base de la table des symboles est définie dans *Table.java*, elle possède un unique attribut, une Map où les clés sont des String correspondant aux noms des fonctions qui sont les branches à la racine de l'arbre de syntaxe abstrait. Les valeurs sont, quant à elles, des *Tuple<Set<String>,Set<String>,List<String>>*, où le premier élément définit les variables en input des fonctions, le second définit les variables déclarées tout au long des fonctions, et le dernier définit les variables en output des fonctions.

Pour ce qui est du parcours de l'arbre, et donc indirectement la construction de la table des symboles, nous commençons par créer un objet *Visitor* auquel nous donnons l'arbre, et qui va initialiser une table vide dans son attribut *« root »* pour représenter la table des symboles correspondant à l'arbre fourni. Nous avons ensuite une méthode analyse dans le fichier *Visitor.java*, qui va s'occuper de séparer les branches (les fonctions) et d'appeler une autre méthode *analyseFunction* pour les analyser une par une.

Nous commençons l'analyse de fonction par vérifier s'il n'existe pas déjà de fonction avec le même nom grâce à un appel à *addFunction* sur la table. Si la fonction est déjà déclarée, la méthode renvoie *false* et une exception est levée, sinon elle est ajoutée à la *Map* et le *Tuple* est initialisé. Ensuite, nous récupérons la branche *« INPUT »* dont les fils sont tous les inputs de notre fonction, et dans une boucle nous appelons la méthode *addInput* sur la table pour réaliser une vérification que l'input courant n'ait pas déjà été déclaré, dans quel cas il est ajouté au *Set* des inputs de la table des symboles pour la fonction courante, sinon une exception est levée. Nous faisons ensuite une étape similaire pour les outputs, si ce n'est que nous ne faisons pas de vérification si un output n'a pas déjà été déclaré, car nous n'avons pas jugé problématique de renvoyer plusieurs fois la même variable, c'est d'ailleurs pour cela que seul l'ensemble des outputs est une List.

Nous poursuivons en récupérant le sous-arbre dont l'étiquette est *« COMMANDS »* qui correspond ainsi à toutes les opérations qui sont effectuées dans la fonction en question. Chacune de ces opérations, ou commandes, seront traitées une à une dans une autre méthode *analyseCommand* du Visitor. Cette méthode permet de réaliser des appels récursifs sur la branche en question, tant qu'il y a des commandes à exécuter. En effet, une commande peut avoir l'étiquette *« ASSIGN »*, *« IF »*, *« WHILE »*, *« FOR »* ou *« FOREACH »*, et ces commandes peuvent comporter de nouvelles commandes comme des for imbriqués, des conditions dans des boucles etc…

- Pour s'assurer que le *« IF »* est correctement construit avec la méthode *ifIsCorrect*, nous vérifions dans un premier temps que sa condition est correcte grâce à la méthode *analyseExpression* de *Visitor* que nous détaillerons plus loin. Si la condition est valide, nous pouvons vérifier que les commandes dans le sous-arbre correspondant au *« then »*, et le sous-arbre correspondant au *« else »* s'il est présent, sont correctes, grâce à un nouvel appel à *analyseCommand* pour chacune d'entre elles.

- Pour s'assurer que le *« WHILE »* et le *« FOR »* sont corrects, nous utilisons la même méthode *loopIsCorrect* car ces deux commandes ont la même structure. Nous commençons comme dans *ifIsCorrect* par vérifier si le sous-arbre *« COND »* est correct par un appel à *analyseExpression*, puis si c'est le cas, nous procédons à la vérification de l'ensemble des commandes du sous-arbre *« COMMANDS »* grâce à *analyseCommand*.

- Même principe pour le *« FOREACH »* avec cependant une étape supplémentaire dans la méthode *foreachIsCorrect*. Nous commençons par vérifier que la variable du *« FOREACH »* n'a pas déjà été déclarée, puis si c'est le cas nous continuons par un appel à *loopIsCorrect*.

- Enfin, pour ce qui est de *« ASSIGN »* ici nous nous occupons d'initialiser ou redéfinir des variables existantes. Dans une version précédente, nous avions une table de vérité qui gardait en mémoire la ligne où était définie la variable ajoutée, mais une version simplifiée l'a remplacée afin d'avoir des données concises et explicites pour le code 3 adresses. Pour cette étiquette, il y a une étape de vérification cruciale, afin de nous assurer que les règles autour du langage while sont respectées. En effet, nous avons découpé cet arbre en deux branches distinctes : une branche avec l'étiquette *« VARIABLES »* sur le noeud le plus haut, et une branche avec l'étiquette *« EXPRESSIONS »*. Les variables sont donc ce que nous allons ajouter à la table des symboles pour cette fonction, et les expressions sont les valeurs attribuées à ces variables. Lorsqu'il s'agit d'assigner une variable à une variable, aucun souci ne se pose en *while* : si elle n'était pas définie elle est automatiquement évaluée à *NIL* et par conséquent la variable à assigner vaut également *NIL*. Cependant, ce qu'il est crucial de vérifier est qu'il y ait autant de variables dans la branche *« VARIABLES »* que de valeurs générés par la branche *« EXPRESSIONS »*, notamment dans le cas d'appel à des fonctions. Si les nombres correspondent, toutes les variables sont ajoutées au Set des variables locales pour la fonction courante dans la table, sinon une exception est levée.

Cette étape de validation du programme s'effectue donc ainsi : nous parcourons tous les enfants du nœud *« EXPRESSIONS »* grâce à la méthode *analyseExpression* qui retourne le nombre de valeurs généré par l'expression. Lorsqu'il s'agit d'un nœud avec l'étiquette *« EQU »*, *« CONS »*, *« LIST »*, *« HD »* ou *« TL »*, nous effectuons une analyse récursive de leur sous expression, et lorsqu'il s'agit d'un nœud avec l'étiquette *« VAR »* nous ajoutons la variable au *Set* des variables locales de la table des symboles pour la fonction courante. Dans les deux cas précédents et le cas par défaut, la méthode retourne 1, car une seule valeur est générée, le cas spécial est lorsqu'il s'agit d'un nœud avec l'étiquette *« CALL »*, nous procédons à son analyse par le biais de la méthode *callIsCorrect* qui retournera le nombre d'output de la fonction appelée.

La méthode *callIsCorrect* vérifie dans un premier temps que nous n'essayons pas d'effectuer un appel sur le main, sinon une exception est levée. Ensuite nous utilisons la méthode *findFunction* qui va parcourir notre AST jusqu'à trouver le sous-arbre correspondant à la fonction que nous appelons, cette méthode permet l'utilisation de fonctions avant leur déclaration. Si aucun arbre n'est retourné nous soulevons une exception. Sinon, nous regardons combien d'inputs la fonction en question a besoin en comptant le nombre d'enfants dans son nœud *« INPUT »*. Ensuite nous reprenons l'arbre appelant, celui avec le nœud *« CALL »* et nous comptons le nombre d'inputs donnés grâce à la méthode *analyseExpression* qui va considérer tous les types d'expression possible. Après avoir compté, nous comparons ce nombre avec le nombre d'inputs requis, s' ils correspondent, cette étape est validée alors *callIsCorrect* va renvoyer le nombre d'outputs de la fonction appelée, sinon nous soulevons une nouvelle exception.

III. Code trois adresses

Pour la génération du code trois adresse, le travail s'effectue au sein de la classe Generateur3a. Un objet de cette classe est initialisé avec l'AST et la table des symboles. Le code généré est stocké dans une Deque<String[]> ou chaque tableau correspond à une ligne de code, et chaque élément à un opérateur, un opérande, ou une instruction (le premier élément étant toujours un opérateur ou une instruction). Nous disposons aussi d'un compteur de registre, d'une fonction génératrice de registre, d'un compteur de labels, d'une String stockant le nom de la fonction en train d'être traduite (pour interagir avec la table des symboles), une Queue pour stocker temporairement des registres, et une autre pour les valeurs à assigner (cette Queue contient des objets de type Assign composés de la valeur à assigner, et du nombre de valeur qu'elle contient, utile pour les fonctions à plusieurs retours).

Pour générer le code il suffit d'appeler la méthode *generate* qui ne va s'occuper que d'appeler la méthode *generateRec* avec l'AST, et de renvoyer le code produit. La méthode *generateRec* quant à elle, comme son nom l'indique va s'occuper de générer le code en parcourant récursivement l'AST. Elle contient une grande instruction switch qui va rediriger vers la génération à faire en fonction du nom du nœud courant.

Le jeu d'instruction du code trois adresse est très limité :

- func begin <*nom*> : annonce le début de la fonction dont on donne le nom.
- func end : annonce la fin de la fonction
- parse <*variable*> <*numéro de l'argument*> : uniquement utilisé pour le main, indique quel argument reçu de la ligne de commande parser et dans quelle variable stocker le résultat.
- print <*variable/registre*> : uniquement pour le main, indique quel valeur afficher. Un registre est utilisé dans le cas d'un output non déclaré, pour stocker la valeur nil. Il peut y en avoir plusieurs.
- return <*registre*> : uniquement pour les fonctions autres que le main, indique quelle valeur retourner, il ne peut y en avoir qu'un par fonction. Dans le cas d'une fonction à plusieurs retours, on construit un arbre binaire contenant tous les retours. Par exemple si la fonction en while a l'instruction : write A, B, C ; nous retournons la valeur (cons A (cons B C)).
- store <*variable/registre*> <*valeur*> : indique de stocker la valeur indiquée dans la variable ou le registre indiqué.
- param <*registre*> : indique que le registre servira de paramètre à une fonction, qu'il faut donc l'ajouter à une pile.

- call <*registre*> <*nom de la fonction*> <*nombre de paramètres*> : indique qu'il faut appeler la fonction donnée avec le nombre de paramètres donné récupérés depuis la pile, et stocker le résultat dans le registre donné.
- label <*nom du label*> : donne un label à la ligne courante. Les labels sont composés d'une chaîne de caractère suivie d'un numéro. Les chaînes possibles sont : loop, end_loop, end_if et false.
- goto <*label*> : indique de continuer l'exécution au label donné.
- ifz <*registre*> <*label*> : indique de continuer l'exécution au label donné si la valeur contenue dans le registre est interprétée comme étant nulle.
- cons <*variable/registre res*> <*variable/registre gauche*> <*variable/registre droit*> : indique de stocker dans la variable/registre « res », l'arbre ayant pour fils gauche la variable/registre « gauche », et pour fils droit la variable/registre « droit ».
- equ <*registre res*> <*registre gauche*> <*registre droit*> : indique de stocker dans le registre « res » le résultat de la comparaison entre le registre « gauche » et le registre « droit ».
- hd <*variable/registre res*> <*variable/registre op*> : indique de stocker dans la variable/registre « res » le résultat de l'opération hd sur l'opérande « op ».
- tl <*variable/registre res*> <*variable/registre op*> : indique de stocker dans la variable/registre « res » le résultat de l'opération tl sur l'opérande « op ».
- nop : indique de ne rien faire.

IV. Bibliothèque runtime de While écrite dans le langage cible

Nous avons choisi C++ pour langage cible, en partie pour pouvoir profiter de la redéfinition des opérateurs notamment celui de cast pour convertir les arbres binaires en entier, booléens ou chaînes de caractère facilement, mais aussi parce que C++ est le langage pour lequel nous trouvions le meilleur équilibre entre nos connaissance du langage et la capacité à créer une bibliothèque dans le langage.

La bibliothèque est très simple, nous avons déclaré toutes nos classes et fonctions dans le namespace « whilestd », nous avons la classe virtuelle *BinTree* définie dans BinTree.h dans lequel nous définissons aussi l'alias *BinTreePtr* pour *std::unique_ptr<const BinTree>*, car c'est de cette sorte que nous stockerons les arbres binaires. Cette classe virtuelle ne présente que les opérations de cast en *int*, en *bool*, et en *std::string*, l'opérateur de comparaison, les méthodes hd et tl, la méthode *clone* pour copier un arbre, ainsi que la méthode *pp* pour l'affichage qui n'est utilisée que par l'opérateur << pour faciliter la lecture du code.

Nous avons ensuite la classe *Leaf* représentant les symboles, elle hérite de *BinTree* et a un attribut « symbol » de type *const std::string* représentant sa valeur. La classe *Node*

représente les nœuds. Elle hérite aussi de *BinTree* et a deux attributs de types *const BinTreePtr* représentant ses fils. La valeur « nil » est représentée par un objet *Node* ayant *nullptr* pour ses deux fils. Nous avons le fichier boolean.h ne contenant qu'une fonction permettant la traduction en *BinTree* d'un booléen, utile pour l'opération « equ » dans le code trois adresse. Enfin nous avons la classe statique *Parser* permettant de traduire en *BinTreePtr* les arguments données en entrée du main, c'est-à-dire un entier ou une formule basée sur « cons ». Le parsing lève une exception en cas d'input invalide.

V. Génération de code à partir du code trois adresses

Pour la génération du code (C++) à partir du code trois adresse, le travail s'effectue au sein de la classe Code3atoCpp. Un objet de cette classe est initialisé avec le code trois adresse généré précédemment dans une Deque<String[]>. Cette classe contient aussi trois expression régulières sous forme de String, permettant de reconnaître les lexèmes « Variables » et « Symbole » de la grammaire, ainsi que les registres du code trois adresses. Elle contient aussi deux BufferedWriter, un pour l'écriture du fichier d'en-tête (nécessaire car nous autorisons l'utilisation de fonctions avant leur déclaration), et un autre pour le fichier source. Enfin il contient une Map<String, StringBuilder> dans laquelle seront stockées les corps des fonctions, avant écriture dans le fichier source.

Pour générer le code, il suffit d'appeler la méthode *generate*, celle-ci prend en argument le nom du fichier contenant le code while (obtenu dans le main via les arguments en ligne de commande), ainsi que la table des symboles. S'il y a des fonctions autre que le main dans la table des symboles, on génère un fichier d'en-tête contenant toutes leurs définitions, sinon on ne génère que le fichier source. Le fichier d'en-tête est généré par la méthode *generateHeader* qui n'utilise que la table des symboles. Le code source lui est généré en partie par la méthode *generateFunBodies*, qui va créer les StringBuilder contenant le corps de chaque fonction, puis par la méthode *generateSource* qui va générer les signatures des fonctions en se basant sur la table des symboles, at écrire les corps des fonctions dans le fichier source.

C'est la méthode *generateFunbodies* qui va utiliser à la fois la table des symboles et le code trois adresses pour générer le corps des fonctions. Pour cela, elle dispose d'une String pour stocker le nom de la fonction courante, un Set<String> pour stocker les inputs récupérés dans la table des symboles, un autre Set<String> pour les variables locales (qui sera complété avec les registres), et un Stack<Stirng> pour stocker temporairement les paramètres. Cette méthode ne fait ensuite que itérer sur les lignes du code trois adresse, et de la même manière que la méthode *generateRec* de Generateur3a, elle utilise une instruction switch en fonction de la première valeur de la ligne de code pour générer le code en conséquence.

Pour ce qui est de l'instruction func, le résultat change en fonction de si la fonction courante est le main ou pas, En effet le main est la seule fonction dont la signature n'a pas

encore été générée, on en profite aussi pour lui ajouter un bloc if-else pour vérifier que le bon nombre d'inputs est entrée en ligne de commande, ainsi qu'un bloc try-catch pour gérer les potentielles exceptions levées par le parsing de la bibliothèque runtime. D'une manière générale, à chaque nouveau fun begin, on initialise les variables locales, car C++ contrairement à while ne permet pas l'utilisation de variable non déclarée, et à chaque func end, on enregistre le StringBuilder dans la Map et réinitialise les variables locales à la méthode.

Les autres spécificités sont les suivantes : l'instruction param ne génère rien, elle se traduit juste en l'ajout dans le Stack du registre concerné pour future utilisation ; l'instruction ifz se traduit en boucle while si le label contient la chaîne « loop », en if sinon ; l'instruction label ne se traduit qu'en une fin de bloc si le label contient la chaîne « end », ou en une fin de bloc if et début de bloc else s'il contient « false » ; le nop ne se traduit en rien ; le goto non plus, nous avions initialement pensé qu'il pourrait être utilisé en C++ sans problème, avant d'apprendre que nous ne pouvions pas déclarer de variable après un label et avant le goto vers ce label.

Enfin, la classe Code3aToCpp dispose d'une méthode *format* qui prend en entrée une String correspondant à un lexème « Variables » ou « Symbole » de la grammaire, ou un registre. Dans le cas d'un registre elle ne fait rien, mais sinon elle modifie la chaîne de caractère en une compatible avec C++ pour ce qui est du nommage de variable ou de fonction (on remplace notamment les « ! » et « ? », et on ajoute un « _ » pour éviter les éventuels conflits avec les mots-clés de C++).

VI. Validation du compilateur

Nous avons écrit des codes en while minimalistes pour faire des sortes de « tests unitaires » de notre grammaire et de la génération de l'AST en utilisant le débogueur de Antlrworks. Ces codes couvrent toutes les règles de notre grammaire ainsi que tous les nœuds de l'AST. Pour ce qui est de la partie en Java, nous avons uniquement fait des vérifications manuelles via des prints de la table des symboles et du code trois adresse généré à partir de nos différents codes while. Enfin nous avons écrit des tests unitaires en utilisant la bibliothèque googletest pour tester notre bibliothèque runtime, ces tests couvrent toutes les fonctions/méthodes sur un critère de *Base Choice Coverage*.

Nous sommes donc sûrs d'avoir une grammaire permettant de décrire l'entièreté du langage while correctement, ainsi qu'une bibliothèque runtime permettant d'effectuer toutes les opérations nécessaires au langage while en C++. Pour l'analyse sémantique et la génération de code trois adresse et C++, nous sommes plutôt sûrs d'être capable de détecter toutes les erreurs et de correctement traduire l'entièreté du langage while, mais ne sommes pas certains puisque nos tests restent non-automatisés et des détails auraient pû nous échapper.

VII. Description de la méthodologie de gestion de projet

Dans la première partie, nous avons travaillé sans outils précis : nous étions chacun sur notre ordinateur sur ANTLRWorks, essayant de créer un AST correct. Néanmoins cela n'était pas un travail individuel car les avancées et problèmes étaient partagés.

Dans la suite du projet nous avons tardivement créer un repository Github afin de stocker notre projet et pouvoir régulièrement mettre à jour les avancées des autres. Nous avons également utilisé Discord comme plateforme de communication que ce soit pour nous répartir les tâches ou bien pour nous partager du code rapidement sans passer par Github. Au niveau de l'IDE nous avons tous opté pour VSCode, cela nous a permis de pouvoir partager notre configuration sans problème.

Concernant la répartition des tâches, comme dit précédemment nous avons tous les trois travaillé sur la grammaire et la production de l'AST au niveau d'ANTLRWorks. Yanis et Benjamin ont peaufiné l'AST pendant que Marius s'occupait d'élaborer l'analyse sémantique. Yanis s'est ensuite penché sur la génération du code trois adresses ainsi que sa conversion en C++ et la bibliothèque runtime associée. Benjamin s'est quant à lui occupé des tests.

Conclusion

En conclusion, le démarrage du projet a été compliqué, car nous avons eu des difficultés à mettre en relation la théorie du cours aux différentes phases du projet.

D'autre part, une fois la partie ANTLR terminée, il nous a été difficile de nous répartir les tâches : certaines parties étaient dépendantes d'autres, ce qui nous à forcé à reprendre notre modèle à plusieurs reprises en prenant du temps sur nos emplois du temps personnel.

Néanmoins ces phases de mise en commun et de reprise de code nous ont semblé essentielles à la compréhension globale du projet, ainsi nous n'aurions pas forcément changé notre organisation.

Ce projet a été une bonne opportunité de comprendre l'organisation de la création d'un langage ainsi que l'utilité de toutes les phases de conception et d'implémentation.