



**Université
de Rennes**



Axel Allain
Ariane Nicolas
Ulysse-Néo Lartigaud
Baptiste Amice
Moussa Berthe
G1 SI ESIR2

Rapport de projet TLC Le langage While

Sommaire :

1. Introduction	3
2. Description technique	3
2.1. Architecture du compilateur et de la chaîne de compilation	3
Architecture	3
Chaîne de compilation	4
2.2. Description de l'AST	4
2.3. Génération de code 3 adresses à partir de l'AST	6
1. Construction de la table des symboles	7
2. Vérification de types	7
3. Génération du code 3 adresses	7
2.4. Génération de code cible à partir du code 3 adresses	9
2.5. Bibliothèque runtime de WHILE écrite dans le langage cible	11
3. Description de la validation du compilateur	11
3.1. Méthodologie utilisée	11
3.2. Couverture de test	12
3.3. Ce qui fonctionne et ce qui ne fonctionne pas	12
4. Description de la méthodologie de gestion de projet	13
4.1. Outils utilisés pour la gestion du projet	13
4.2. Étapes de développement / découpage des tâches	14
4.3. Rapport individuel	14
5. Conclusion	16

1. Introduction

Le développement d'un compilateur représente un défi, nécessitant une compréhension approfondie des langages source et cible, ainsi qu'une mise en œuvre efficace des différentes phases du processus de compilation. Ce rapport vise à présenter notre projet de création d'un compilateur pour le langage WHILE en fournissant une vue d'ensemble de notre méthodologie, des outils utilisés, des étapes de développement, et en soulignant les aspects fonctionnels et les défis rencontrés au cours de ce processus.

2. Description technique

2.1. Architecture du compilateur et de la chaîne de compilation

Architecture

L'architecture de notre compilateur est constituée de plusieurs composantes :

Tout d'abord, plusieurs packages viennent constituer le corps du programme de compilation. Le package antlrworks contient le parser et le lexer générés par antlrworks. Le package compilateur contient notre visiteur abstrait pour parcourir l'AST¹, des visiteurs pour construire la table des symboles, vérifier la syntaxe et construire le code 3 adresses, ainsi qu'un traducteur du code 3 adresses vers C et finalement une exception de compilation personnalisée pour connaître l'erreur, ainsi que la ligne et le fichier concernés. Un fichier Main permet de gérer et connecter ces différentes composantes. Le diagramme de classes (simplifié) illustre les éléments constituant la composante Java du compilateur :

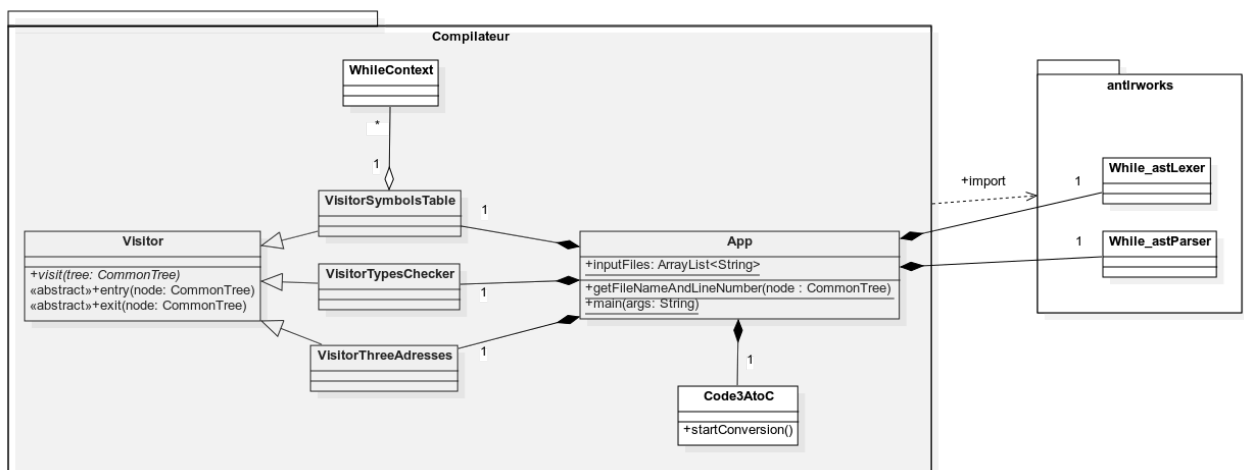


Diagramme de classes simplifié de la partie Java du compilateur

Pour la traduction du langage while en langage C, une librairie spécialisée a été créée. Elle se compose notamment d'une implémentation des arbres du langage while.

¹ Arbre de syntaxe abstraite

Chaîne de compilation

La chaîne de compilation d'un programme while est donc la suivante :

1. L'AST du programme est construit;
2. La table des symboles du programme est construite à partir de l'AST;
3. Une vérification des types est effectuée à partir de l'AST et de la table des symboles;
4. Le code 3 adresses est généré à partir de l'AST et de la table des symboles;
5. Le code 3 adresses est traduit en code C, en se basant sur notre librairie spécialisée;
6. la compilation du code C.

2.2. Description de l'AST

Nous allons maintenant vous décrire notre AST ou Arbre de Syntaxe Abstrait nous permettant de stocker les lexèmes pertinents, d'encoder la structure grammaticale sans information superflue et qui est facilement manipulable pour la suite des étapes. Il est différent de l'arbre de dérivation syntaxique, résultant de notre grammaire, qui représente la manière dont les règles de grammaire du langage ont été appliquées pour générer la séquence de lexèmes dans le programme. L'AST se concentre sur la signification sémantique du programme, tandis que l'arbre de dérivation syntaxique se concentre sur la manière dont le programme est syntaxiquement construit. Un AST est plus simple, et possède des tokens abstraits, utilisés pour améliorer la compréhension du découpage du programme.

Ci dessous un exemple d'AST construit à partir d'un programme While :

```
// Add function
function add :

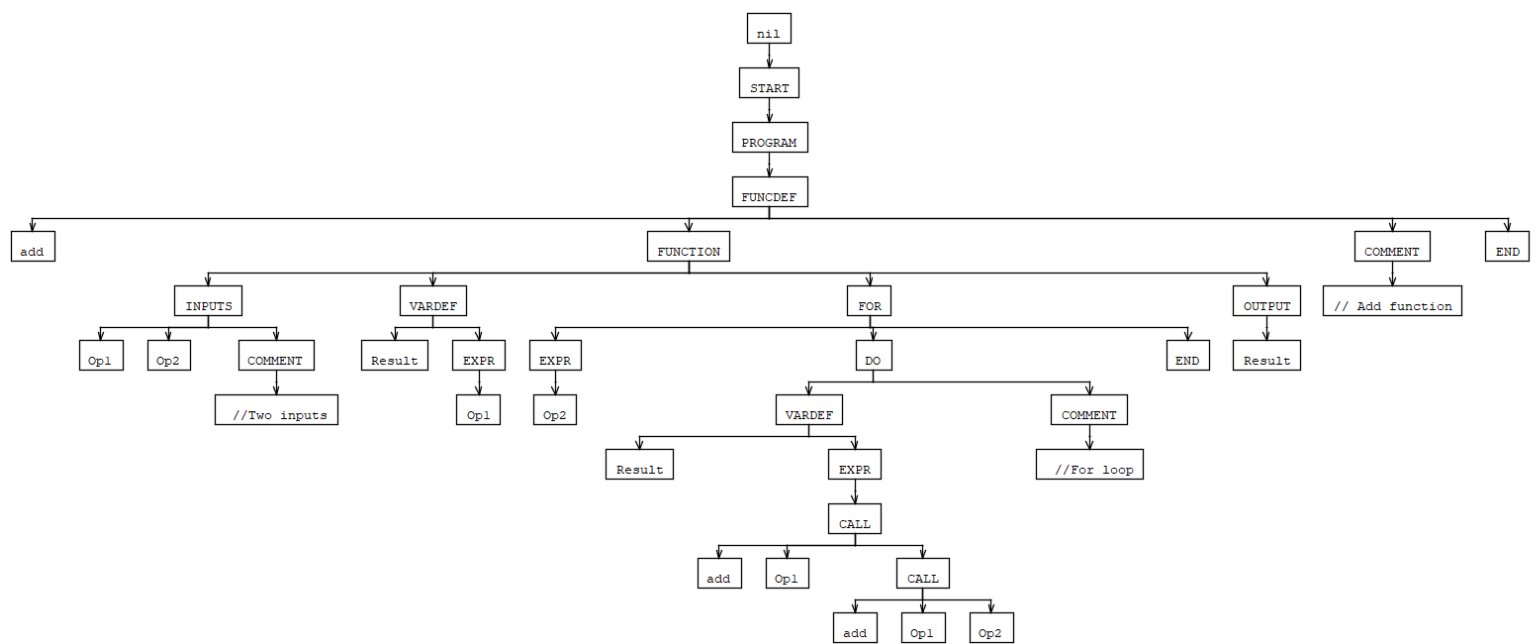
read Op1, Op2 //Two inputs
%

Result := Op1 ;

for Op2 do //For loop
  Result := (add Op1 (add Op1 Op2))
od //Out of the do
%

write Result
```

Exemple de programme While



AST de l'exemple précédent de programme While

Voici les différents tokens que nous avons créés afin de faciliter la compréhension de notre arbre :

- **Programme (START) :**

Racine de l'AST représentant l'ensemble du programme.

Contient les définitions de fonctions et les instructions du programme.

- **Définition de Fonction (FUNCDEF) :**

Noeud représentant la définition d'une fonction.

Comprend le nom de la fonction, les paramètres et le corps de la fonction.

- **Instructions (PROGRAM) :**

Représente une séquence d'instructions dans le programme.

Contient des commandes telles que la lecture d'entrée, l'exécution de commandes, et l'écriture de sortie.

- **Lecture d'Entrée (INPUTS) :**

Noeud représentant l'instruction de lecture d'entrée.

Comprend les variables qui stockent les valeurs lues.

- **Écriture de Sortie (OUTPUT) :**

Noeud représentant l'instruction d'écriture de sortie.

Contient les variables dont les valeurs seront écrites en sortie.

- **Déclaration de Variable (VARDEF) :**

Noeud représentant la déclaration de variables avec une affectation.

Inclut les noms des variables et leurs valeurs initiales.

- **Conditionnel (IF) :**

Noeud représentant une structure conditionnelle "*if-then-else*".

Comprend une condition, le bloc d'instructions à exécuter si la condition est vraie, et éventuellement un bloc d'instructions pour le cas "*else*".

- **Boucle While (WHILE) :**

Noeud représentant une boucle "*while*".

Contient une condition et le bloc d'instructions à répéter tant que la condition est vraie.

- **Boucle For (FOR) :**

Noeud représentant une boucle "*for*".

Comprend une condition de boucle et le bloc d'instructions à répéter.

- **Boucle ForEach (FOREACH) :**

Noeud représentant une boucle "*foreach*".

Contient la variable de l'itération, la liste sur laquelle itérer, et le bloc d'instructions à répéter.

- **Expression (EXPR) :**

Représente une expression dans le langage.

Peut inclure des opérations, des variables, des constantes, des appels de fonctions, etc.

- **Appel de Fonction (CALL) :**

Noeud représentant l'appel d'une fonction.

Contient le nom de la fonction appelée et les arguments passés.

- **Liste (LIST) :**

Représente la création d'une liste.

Comprend les éléments de la liste.

- **Opérations sur les Listes (CONS, HD, TL) :**

Noeuds représentant les opérations de cons, de tête (head), et de queue (tail) sur les listes.

- **Symbole (Symbol) :**

Représente un symbole ou une variable dans le programme.

- **Commentaires (COMMENT) :**

Noeud représentant les commentaires dans le code source.

2.3. Génération de code 3 adresses à partir de l'AST

Après avoir finalisé notre AST, nous sommes passés à l'étape du code intermédiaire, ou code trois adresses. L'utilité de ce code est de représenter notre programme sous forme

d'instructions simples, qui seront facilement traduisibles dans le langage cible par la suite. Pour construire notre code trois adresses, nous avons dû réaliser plusieurs étapes : Tout d'abord, il nous fallait développer un moyen de parcourir facilement notre AST pour en retirer toutes les informations nécessaires.

Nous avons donc commencé par implémenter une classe abstraite **Visiteur**, qui contenait une fonction *visit()*, permettant de parcourir notre arbre en profondeur de manière récursive. Nous avons aussi décidé de créer des fonctions abstraites *entry()* et *exit()*, non implémentées, qui permettent aux classes concrètes de visiteurs d'effectuer leurs actions durant le parcours. Celles-ci sont donc appelées à chaque nœud de l'arbre. Information importante, *exit()* est appelée après les appels récursifs de la fonction *visit()* sur les différents nœuds fils. Cela permet ainsi de s'assurer que les nœuds fils aient déjà été visités, et donc de remonter de l'information aux nœuds parents.

Afin de faciliter la traduction de notre AST jusqu'au langage cible, nous avons besoin de trois passes sur notre AST :

- Une passe pour créer la **Table des symboles**,
- Une passe pour la **Vérification de types**,
- Et enfin une passe pour la construction du **code 3 adresses**.

Nous avons donc créé trois classes distinctes de *Visitor* pour nous faciliter le travail.

1. Construction de la table des symboles

La table des symboles constitue notre base pour nous repérer dans les différents blocs de code, et pour retrouver facilement nos variables et nos fonctions dans le programme. L'objectif de cette passe est donc de sauvegarder chaque variable ou fonction que nous rencontrons, ainsi que les différents contextes du code source.

Dans notre implémentation, la table des symboles est une liste de contextes du langage While. Pour représenter un contexte du langage While (correspondant à une fonction), nous avons créé une classe comprenant un nom, des variables, des inputs, des outputs. Le visiteur de la table des symboles se charge donc de la construire en parcourant les différents nœuds de l'AST :

- nœud FUNCDEF: création d'un contexte et changement de contexte courant
- nœud INPUTS : ajout des valeurs textuelles des enfants du nœud en input du context courant.
- nœud VARDEF: ajout de valeur textuelle du nœud aux variables du context courant.
- nœud OUTPUT : ajout de valeur textuelle du nœud en output du context courant.

2. Vérification de types

Le typage du langage While est défini par le nombre de valeurs qu'un élément du programme contient. Par exemple, une variable contenant deux arbres n'aura pas le même type qu'une variable contenant un arbre. Par contre, elle aura le même type que deux variables contenant un arbre chacune. La vérification de type consiste donc à vérifier le nombre de valeurs contenues dans des nœuds concernées par des opérations d'affectation, de comparaison, en entrée et sortie de fonction, ou encore en paramètre d'instruction du langage telles que des boucles. D'autres précautions plus mineures sont à prendre, telles que la vérification de déclaration des fonctions pour leur appel. La rencontre d'une erreur de typage arrête la compilation et affiche l'erreur et la ligne concernée du programme.

3. Génération du code 3 adresses

La troisième passe sur notre AST est réalisée par notre *VisitorThreeAdresses* afin de générer le code trois adresses correspondant à notre code source.

Ce visiteur utilise uniquement la fonction *exit()*, car il remonte des informations vers les nœuds parents et non l'inverse. On parcourt donc notre AST en commençant par les feuilles, et pour chaque nœud on associe un **attribut**, qui correspond à la liste des codes 3 adresses générés par les nœuds fils, à laquelle on concatène le code 3 adresses généré par le nœud actuel. Le nœud racine contient donc le code 3 adresses total du programme.

Nous avons décidé de créer nos lignes de code 3 adresses sous la forme suivante :

```
public class ThreeAdresses {  
    public String op;  
    public String arg1;  
    public String arg2;  
    public String var;  
};
```

op : l'opérateur de la ligne, c'est à dire l'action à effectuer

arg : argument, qui peut être un paramètre d'entrée, une valeur de retour...

var : valeur qui vient préciser l'opérateur (nom de la fonction appelée, nom du bloc où on entre...)

Nous avons une petite liste d'opérateurs correspondants à un certain nombre d'actions, qui fonctionnent de la façon suivante :

IGNORE, null, null : ligne à ignorer, correspondant à un commentaire ou à une information inutile pour le code cible.

READ, null, input : indique que la variable input est un paramètre de la fonction courante.

PARAM, variable, null : la variable en argument 1 est le paramètre de la fonction qui sera appelée par la suite.

CALL, RegistreDeRetour, nomFonction : appelle la fonction nomFonction et stocke le résultat dans RegistreDeRetour s'il s'agit d'une fonction à un seul retour (tl, cons...). Pour les fonction à plusieurs retour, RegistreDeRetour contient une string où sont concaténés les noms d'autant de registres que d'outputs nécessaires (ex : "Reg_1 Reg_2 Reg_3"), et l'attribution à chacun des registres se fera dans la traduction en langage cible.

ENTER, null, nomBloc : indique l'entrée dans un nouveau bloc ou dans une nouvelle fonction.

ASSIGN, null, variableAAssigner : stocke la valeur à assigner pour la prochaine définition de variable. Il peut s'agir d'un nom de variable ou d'un registre de retour d'une fonction.

ASSIGNED, null, variableAssignée : stocke le nom de la variable à assigner.

GOTO_IF_NOT_NIL, variableAVérifier, blocOùAller : indique qu'il faut vérifier si la variable est égale à nil (à l'aide de la bibliothèque du langage cible), et aller au bloc indiqué si c'est le cas..

GOTO_IF_NOT_TRUE, variableAVérifier, blocOùAller : indique qu'il faut vérifier si la variable est fausse (à l'aide de la bibliothèque du langage cible), et aller au bloc indiqué si c'est le cas.

exemple de code 3 adresses :

<pre>function main : read Op1, Op2 % Result := Op1 ; for (tl Op2) do Result := (cons nil Result) od % write Result</pre>	<pre>ENTER main FUNCTION READ null Op1 READ null Op2 ASSIGNED null Result ASSIGN null Op1 IGNORE null null PARAM Op2 null CALL Reg_0 tl GOTO_IF_NIL Reg_0 block1 ENTER null block0 IGNORE null null PARAM nil null PARAM Result null CALL Reg_1 cons ASSIGNED null Result ASSIGN null Reg_1 PARAM Reg_0 null CALL Reg_0 tl GOTO_IF_NOT_NIL Reg_0 block0 ENTER null block1 ENDFUNC null null</pre>
--	--

2.4. Génération de code cible à partir du code 3 adresses

Une fois notre code 3 adresses bien terminé, il nous suffit de le parcourir ligne par ligne et de traduire la ligne en langage cible, et ce pour n'importe quel langage cible (normalement).

Nous avons décidé de choisir le C comme langage cible, car ce n'est pas un langage objet, comme le While, et qu'il permet l'utilisation des goto, ce qui nous est utile pour traduire depuis le code 3 adresses. La traduction se fait en écrivant le code dans un fichier output.c et la signature des fonctions dans un fichier output.h.

On parcourt donc le code le code 3 adresses, et en fonction de l'opérateur identifié, on écrit dans les fichiers les informations correspondantes.

Les étapes les plus complexes sont les suivantes :

- La signature d'une fonction, reconnue par l'opérateur ENTER FUNCTION, est écrite à l'aide de la table des symboles : on récupère le nombre d'inputs et d'outputs de la fonction. En C, il n'est pas possible de créer des fonctions à plusieurs valeurs de retour, nous allons donc fonctionner avec les pointeurs : toutes les fonctions seront de type void, et nous passerons en paramètre les pointeurs sur les arbres qui seront retournés à la fin. On ajoute ensuite tous les inputs indiqués par la table des symboles :

```
function test_return :  
  read Op1, Op2  
  %  
  Result := Op1 ;  
  %  
  write Result
```

La fonction ci-dessus aura comme signature :

```
public void test_return(Tree* Result, Tree Op1, Tree Op2);
```

- Les appels de fonction se font donc de la même manière : on commence par créer autant d'arbres qu'il faut de registres d'output (récupérés en parsant la string de retour de l'opérateur CALL), puis on appelle la fonction voulue avec en paramètres ses valeurs d'output, puis ses valeurs d'input.
- Les valeurs d'inputs sont récupérées à l'aide de l'opérateur PARAM, et stockées dans une liste. Lorsqu'on reconnaît l'opérateur CALL, on vide la liste d'autant d'input que nécessaire, en commençant par la fin pour être sûrs qu'il s'agit bien des paramètres de la bonne fonction.

La ligne : `Result := (add Op1 Op2)` en While sera traduite en C de la façon suivante :

```
Tree Reg_i = nil;  
add(&Reg_i, Op1, Op2);  
*Result = Reg_i;
```

On remarque que add() prend bien en paramètre le pointeur vers Reg_i, suivi de ses deux paramètres Op1 et Op2. On assigne ensuite à Result la valeur retournée par add(), à savoir Reg_i.

- La gestion des égalités multiples a aussi été un point de réflexion car impossible à réaliser en C. Nous avons opté pour une liste de variables à assigner et une liste de variables assignées. Lorsque la définition de variables est terminée (ie lorsque l'opérateur suivant n'est plus ASSIGN), on parcourt les deux listes et on les associe. Si la variable à assigner est en réalité une fonction, on récupère ses registres de sortie pour les ajouter dans la liste.

2.5. Bibliothèque runtime de WHILE écrite dans le langage cible

La bibliothèque runtime de WHILE écrit en C permet de faciliter sa traduction vers C en mettant à disposition les fonctions et concepts propres au WHILE nécessaires à la conversion.

Pour réaliser cette conversion, il est essentiel de représenter les arbres binaires du WHILE en langage C car c'est le concept central du langage WHILE. Pour cela nous avons créé une structure Tree contenant un pointeur sur un Tree faisant office de fils gauche (head) et un autre faisant office de fils droit (tail). Cette structure contient aussi une valeur sous forme de chaîne de caractères.

Ainsi, cette structure permet la représentation d'un arbre binaire en langage C. Nous avons créé un set de fonction permettant de manipuler ces arbres en fonction des besoins du langage WHILE :

- **cons**, permet de créer un arbre binaire et alloue l'espace mémoire nécessaire.
- **deleteTree**, permet de supprimer un arbre binaire et libère son espace mémoire.
- **displayString**, parcourt l'arbre et affiche la valeur des nœuds de l'arbre.
- **pp**, affichage intelligent des arbres en fonction du type (int, bool, string) de l'arbre.
- **isEmpty**, vérifie si un arbre est vide.
- **isLeaf**, vérifie si un arbre est une feuille.
- **copy**, fait une copie d'un arbre.
- **equals**, vérifie si deux arbres sont égaux.

En WHILE il n'y a pas de concept de type comme en C. Il faut donc convertir les arbres en fonction des types du langage C :

- **boolTree**, convertit un arbre en booléen.
- **intTree**, convertit un arbre en int.

En WHILE il est possible de passer un arbre en argument lors de l'exécution. Il faut donc convertir les arguments en entrée du programme en arbre :

- **parsArgs**, convertit un argument en arbre.
- **buildTreeByInt**, construit un arbre représentant un int.
- **buildTreeByString**, construit un arbre à partir d'un string.

Il est aussi possible de faire l'opération inverse avec la fonction **buildStringFromTree** c'est-à-dire de construire un string en fonction d'un arbre.

3. Description de la validation du compilateur

3.1. Méthodologie utilisée

L'installation des technologies de test s'est faite par le biais de Maven. La suite de tests a été écrite en utilisant la librairie de tests unitaires JUnit. Nous avons pu l'utiliser pour réaliser des tests unitaires, mais également - de manière moins conventionnelle - pour tester individuellement les fonctionnalités implémentées du langage While sur l'ensemble de la chaîne de compilation. Ainsi, en utilisant Surefire, la suite de test est exécutée lors du build de l'application. La classe testApp permet de tester la chaîne de compilation, ainsi que les différentes fonctions et fonctionnalités du langage.

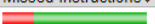
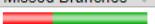
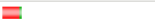
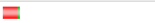














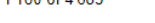
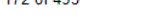
L'objectif est de vérifier si le sous-ensemble du langage a été correctement implémenté.

3.2. Couverture de test

Pour tester notre compilateur, nous avons réalisé des tests sur les différentes spécifications du langage While. Nous avons par exemple écrit des tests permettant de tester les fonctionnalités individuelles du langage telles que les if et les for, mais aussi des tests pour des programmes plus complexes. Nous avons également écrit des tests de résilience du programme, par exemple face à des commentaires sur toutes les lignes d'une fonction, ou des espaces aléatoires. Il a également fallu rédiger des tests d'échec sur des programmes invalides. Cela nous a permis d'effectuer une certaine intégration continue pendant notre développement, si nous modifions quelque chose dans notre code, nous vérifions grâce aux tests que ce que nous avons codé auparavant fonctionne toujours.

Nous avons aussi implémenté des tests unitaires pour la table des symboles afin de vérifier les différents contextes dans un programme. Les contextes permettent de comprendre d'où viennent les variables ce qui est important lors de l'utilisation des variables d'entrées (inputs) et de sorties (outputs).

Le test des différentes fonctionnalités du langage While, nous a permis de grandement tester la logique de notre code et ses embranchements, ainsi que d'avoir un test de coverage satisfaisant. Ci-dessous, on peut voir un rapport de tests Jacoco :

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
VisitorThreeAdresses		78 %		65 %	75 139	71 369	1 7	0 1
App		16 %		16 %	15 19	62 73	2 4	0 1
code3AtoC		79 %		69 %	30 81	31 204	2 13	0 1
VisitorTypesChecker		82 %		71 %	11 29	10 81	0 4	0 1
VisitorSymbolsTable		85 %		75 %	11 31	5 59	1 9	0 1
WhileContext		86 %		100 %	1 11	1 19	1 10	0 1
App.new_OutputStream().{...}		0 %		n/a	2 2	2 2	2 2	1 1
WhileException		66 %		n/a	0 1	2 4	0 1	0 1
Visitor		100 %		80 %	2 8	0 14	0 3	0 1
VisitorThreeAdresses.ThreeAdresses		100 %		n/a	0 1	0 1	0 1	0 1
Total	1 160 of 4 683	75 %	172 of 493	65 %	147 322	183 825	9 54	1 10

Rapport de tests Jacoco

Ce rapport nous permet d'identifier la logique non testée de l'application, ainsi que le code mort. On s'aperçoit que le code du fichier main n'est que peu testé, ce qui est parfaitement normal, le contexte d'un test unitaire pouvant être considéré comme sa propre fonction main.

Si nous avons eu le temps, il aurait été préférable d'également réaliser une table de partition et des tests de mutation.

3.3. Ce qui fonctionne et ce qui ne fonctionne pas

Ce qui fonctionne dans notre compilateur :

- La lecture de fichiers While;
- La création d'un AST à partir de ces fichiers;
- La création de la table des symboles;
- La vérification syntaxique et l'affichage du fichier et de sa ligne à l'origine d'une l'erreur;
- La création d'un code 3 adresses;
- La traduction du 3 adresses en C;

- La compilation de ce code C.

Ce qui ne fonctionne pas dans notre compilateur :

Nous pensons que la manière dont nous avons géré les fonctions à plusieurs retours dans notre code 3 adresses n'est pas forcément adaptée, il ne serait pas possible de la traduire en assembleur par exemple car il faut parser une string pour obtenir le nombre exact de retour de la fonction.

Nous n'avons pas non plus eu le temps de réaliser la partie optimisation, qui pourtant aurait été intéressante à coder.

4. Description de la méthodologie de gestion de projet

4.1. Outils utilisés pour la gestion du projet

Le développement du compilateur du langage While a été mené à bien grâce à l'utilisation d'une série d'outils de gestion de projet. Ces outils facilitent la collaboration, la communication et le suivi des différentes étapes du processus de développement.

Gitlab a été central dans notre gestion de versions tout au long du projet. Cette plateforme de gestion de code source a permis un suivi des modifications apportées au code, favorisant la collaboration entre les membres de l'équipe. Les fonctionnalités de suivi des problèmes et de demandes de fusion ont également facilité la résolution des conflits et le maintien d'une base de code stable.

La communication efficace étant importante dans un projet collaboratif, Discord a été notre principal canal de communication. Les canaux textuels et vocaux ont permis le partage d'idées entre les membres de l'équipe et ont également contribué à maintenir une communication réactive même pendant les vacances.

Ensuite, Trello a été utilisé comme tableau de gestion de projet visuel, offrant une vue d'ensemble des tâches à accomplir par chaque personne et de leur statut. Les tableaux Trello ont reflété les différentes phases du développement du compilateur, offrant ainsi une vision claire de l'avancement global du projet.

VSCoDe LiveShare a grandement amélioré notre capacité à collaborer sur le code source à plusieurs en même temps. Cette extension pour Visual Studio Code a permis aux membres de l'équipe de partager leur environnement de développement, facilitant la résolution conjointe de problèmes, la revue de code et la correction d'erreurs.

L'intégration de ces outils a joué un rôle essentiel dans la gestion réussie du projet, en favorisant une collaboration efficace, une communication et un suivi rigoureux des différentes phases de développement.

4.2. Étapes de développement / découpage des tâches

1. Division en 2 groupes (Baptiste, Ulysse, Moussa & Ariane, Axel)
2. Analyse du langage while
3. Conception du diagramme de classe Java (Main, Visiteurs, Traduction code 3A...)
4. Création de la grammaire → Obtention d'un arbre de dérivation syntaxique
5. Génération de l'AST
6. Création syntaxe
7. Ecriture du visiteur abstrait
8. Ecriture du visiteur constructeur de la table des symboles
9. Ecriture du visiteur type checker
10. Ecriture du visiteur constructeur du code 3 adresses
11. Traduction du code 3 adresses en C
12. Ecriture du programme java principal (lecture fichier param, etc.)
13. Tests
14. Déploiement (maven, etc.)

4.3. Rapport individuel

Baptiste :

J'ai travaillé pendant la première moitié du projet en trio avec Ulysse-Néo et Moussa. Ensemble, nous avons conçu le diagramme de classes. Nous avons également écrit le visiteur de construction de la table des symboles (VisitorSymbolsTable). Nous avons aussi écrit un système d'affichage des erreurs (utilisant l'héritage des exceptions Java). Nous avons, de plus, fait en sorte de récupérer le contenu des fichiers passés en paramètres du programme pour la compilation. Nous avons démarré la réflexion sur l'implémentation du typeChecker, puis je l'ai développé. Après cela, j'ai commencé à travailler individuellement sur des tâches plus annexes du projet. Tout d'abord, j'ai rédigé les tests unitaires du compilateur et ceux automatisés du langage While par l'utilisation de Junit. Bien que les tests du langage While ne soient pas réellement des tests unitaires, ils s'en rapprochent, en ce qu'ils testent des fonctionnalités atomiques du langage à compiler. J'ai donc considéré la librairie de test comme adaptée. J'ai ensuite pu migrer le projet vers un projet maven pour automatiser ces tests lors du build du projet et également pouvoir générer des rapports de tests, une documentation javadoc et créer une archive jar exécutable. J'ai d'ailleurs écrit la première moitié du script de compilation utilisant ce jar. J'ai également écrit la méthode permettant d'obtenir le fichier source et la ligne du fichier pour un nœud de l'AST. J'ai donc pu la coupler avec notre système d'affichage d'erreurs. De plus, j'ai ajouté le mode verbeux du compilateur, permettant de choisir si l'on souhaite afficher en console ou non les détails de la compilation. Je me suis également occupé de la plupart des merges de branches Git du projet. Finalement, j'ai rédigé une grande partie du fichier README.md et quelque peu participé à la rédaction de ce rapport. Je n'ai toutefois pas été très présent sur les 3 derniers jours du projet (car en stage). Je remercie donc mes collègues pour le travail qu'ils ont pu fournir sur cette période pour achever le projet.

Moussa :

Dès le début du projet, Ulysse-Néo, Baptiste et moi devions nous occuper du diagramme de classes, de la table des symboles, du typeChecker, puis de traduire certaines parties en langage C. J'ai travaillé pendant ce projet en trio avec Ulysse-Néo et Baptiste. Nous avons commencé par concevoir le diagramme de classes pour faire ressortir les différentes classes qui intervenaient. Par la suite, nous avons pensé à utiliser le design pattern 'visitor' pour parcourir l'AST. Aussi, pour construire la table des symboles, nous avons écrit le visiteur (VisitorSymbolsTable). Nous avons, de plus, fait en sorte de récupérer le contenu des fichiers passés en paramètres du programme pour la compilation. Avec Ulysse-Néo et Baptiste, nous avons démarré la réflexion sur l'implémentation du typeChecker durant les deux dernières séances de TP, et Baptiste l'a développé. Pendant les vacances, j'ai participé à une réunion en visioconférence sur Discord, au cours de laquelle Ariane et Axel nous ont présenté l'avancement du code 3 adresses. Mon implication moins importante dans les dernières parties par rapport à celle de mes collègues est due au fait que je ne maîtrise pas le langage C et aussi au grand nombre de projets que nous avons en SI.

Ulysse-Néo :

Dans le cadre de ce projet, j'ai travaillé avec Baptiste et Moussa sur plusieurs aspects du développement du compilateur. Comme mentionné précédemment, au début du projet, nous avons fait la modélisation de celui-ci ainsi que conçu le visiteur pour générer la table des symboles, le type checker et le code à trois adresses. Après avoir créé le visiteur, avec Baptiste et Moussa nous avons créé la class VisitorSymbolsTable permettant de générer la table des symboles. Après avoir développé la class VisitorSymbolsTable, j'ai travaillé sur le début de l'implémentation du typeChecker toujours avec Baptiste et Moussa mais Baptiste a pris ensuite le relais sur cette partie car je me suis consacré pleinement à la création en de la bibliothèque runtime en C. Pour cette phase du projet, j'ai travaillé en collaboration avec Ariane. Pendant qu'elle faisait la conversion du code à trois adresses en C, je créais la bibliothèque pour que ces deux parties du projet coïncident afin que la conversion du code à trois adresses se passe pour le mieux.

Bien que ma contribution ait été plus marquée dans certaines parties du projet comme la modélisation, le visiteur, la table des symboles ou la bibliothèque C. J'ai également participé à la rédaction de certains aspects du rapport. Nous avons su surmonter tous les défis auxquels nous avons été confrontés. J'ai trouvé l'ambiance de l'équipe de projet très agréable car tout le monde était très réactif et motivé.

Ariane :

Durant presque tout le projet nous avons scindé le groupe en deux, j'ai pour ma part travaillé en binôme avec Axel. Nous avons commencé par réécrire au format antlr la grammaire fournie, puis la transformer en AST compréhensible pour la traduction en code 3 adresses. Nous avons ensuite implémenté la classe VisitorThreeAdresses, qui parcourt l'AST et en ressort le code 3 adresses correspondant. Travailler à deux a été un réel avantage, surtout pour comprendre les notions nouvelles que nous manipulions. Nous avons décidé ensemble quels opérateurs utiliser pour le code 3 adresses, comment faire pour récupérer de l'information des nœuds fils etc... Cependant dans une première partie nous avons fonctionné sans goto (en implémentant uniquement un opérateur ENTER FOR/ END

FOR par exemple), et quand nous avons compris l'intérêt du code trois adresses à savoir être traduisible dans n'importe quel langage, je me suis chargée de refactoriser notre code. Comme nous avons pris du temps pour comprendre les notions du projet, nous n'étions pas très avancés à la fin des TP, il a fallu se répartir les tâches à distance, et je me suis chargée en quasi totalité de la traduction vers le C, avec l'aide d'Axel sur quelques points. Mais je n'ai pas trouvé cela trop pesant car toute l'équipe était très réactive pour répondre à mes questions, et chacun a beaucoup travaillé de son côté pour fournir la librairie C, les tests, le script de compilation, le README... En résumé j'ai trouvé le projet très intéressant, et la dynamique de groupe agréable malgré le sujet parfois fastidieux. J'aurais aimé avoir plus de temps pour appréhender les notions, car je pense que si je devais refaire ce compilateur je prendrais beaucoup moins de temps.

Axel :

Pour le projet de TLC, j'étais en duo avec Ariane lors des séances de TP et lors de séances hors TP. Notre duo s'est tout d'abord occupé d'écrire la première grammaire de base et de construire un premier AST pendant les premières séances de TP. En parallèle, j'ai commencé à rédiger et structurer notre rapport, notamment la structure et la logique derrière l'AST ainsi que les outils utilisés. Ensuite, après que l'autre groupe ait implémenté le visiteur de l'arbre, notre duo a commencé à travailler sur le code 3 adresses. En séance de TP, nous avons commencé à rédiger une première version reconnaissant les programmes simples avec un AST de petites tailles et peu complexe. Nous avons ensuite travaillé pendant les vacances pour continuer à avancer sur le projet. Ariane a travaillé plus en profondeur sur le code 3 adresses et moi-même sur la grammaire en ajoutant certaines fonctionnalités que nous avons oubliées ou mal implémentées. Pour la grammaire c'était l'ajout des commentaires, la simplification de certaines parties de l'AST ou la succession de fonctions dans un programme. Nous avons travaillé de paires en vocal Discord avec des outils comme Visual Studio Live Share pour pouvoir faire du Dual Programming. À la rentrée, Ariane a commencé à travailler sur la traduction du code 3 adresses en C et a beaucoup travaillé dessus. Je l'aidais quand elle travaillait dessus en présentiel mais de mon côté, j'essayais de comprendre les erreurs que notre AST provoque dans son code et ajuster notre grammaire en conséquence. J'ai aussi réalisé quelques programmes de tests compliqués (imbrication de for et if, plusieurs fonctions, appels de fonctions) permettant de tester le programme de mon côté. Mon investissement moins important qu'Ariane sur la traduction du code 3 adresses en C est dû à d'autres projets à rendre pendant la même période et où je me suis investi. A la fin, j'ai participé à la rédaction du rapport final et au déploiement final du compilateur.

5. Conclusion

En conclusion, la réalisation de ce compilateur pour le langage WHILE a représenté un parcours stimulant, marqué par la nécessité d'une compréhension approfondie des nuances du langage source et par la mise en œuvre précise des différentes étapes du processus de compilation.

Ce rapport offre un aperçu détaillé de notre parcours dans la création d'un compilateur WHILE. Nous sommes fiers des réussites obtenues malgré beaucoup de difficultés rencontrées. Ce projet a été une opportunité d'apprentissage significative pour notre équipe, renforçant nos compétences dans le domaine de la compilation.