ECOLE SUPERIEURE D'INGENIEURS DE RENNES

UNIVERSITÉ DE RENNES 1

Rapport de fin de projet

Compilation

Equipe C#

Equipe projet :

Florent Catiau-Tristant (Chef de projet)

Dylan Béhêtre (Responsable des tests)

Yoann Boyère

Alexis Brault

Mehidine Chupeau

Quentin Olivier

Année 2016 - 2017

Sommaire

Table des matières

ECOLE SUPERIEURE
D'INGENIEURS DE RENNES

UNIVERSITÉ DE
RENNES 1

Année 2016 - 2017

Introduction

Ce rapport contient les informations techniques et relatives à la gestion du projet de compilation. Ce projet de compilation entre dans le cadre du module de Compilation du semestre 7 de la formation d'ingénieurs de l'ESIR en spécialité Technologies de l'Information, option Systèmes d'Information.

L'entièreté du projet est disponible sur github. Le projet est ainsi disponible et visible par tous.

I. Description technique

Notre équipe a opté, lors du démarrage du projet, pour un compilateur vers le langage C#. Nous avons fait ce choix pour des questions d'affinités avec le langage et de découverte. Bien que celui-ci possède de fortes ressemblances avec le langage Java, il dispose de quelques particularités que nous souhaitions explorer.

1. Schémas d'exécution

Notre implémentation est basée sur un modèle général de traduction de code. Typiquement, le passage des paramètres pour chaque appel de fonction, quelle qu'elle soit, se traduit par un couple de files. Nous utilisons très peu les fonctionnalités présentes dans C#. Cela dit, le fonctionnement du langage ne permet pas vraiment une traduction simplifiée, en se passant de la méthode générale utilisée.

Nous détaillerons dans la suite le format de traduction pour chaque partie importante du langage WHILE.

a. Passage de paramètres et appels de fonctions

Chaque appel à une fonction (locale ou d'une librairie) induit de passer des paramètres. Afin de gérer le passage de paramètres multiples d'une manière commune, nous utilisons deux files : `Queue<BinTree> inParams` et `Queue<BinTree> outParams`. Ainsi, nous n'avons pas à gérer le nombre de paramètres : dès qu'il y a un paramètre, on l'ajoute à la file avec la méthode `Enqueue()`. L'opération inverse permet de récupérer ces paramètres dans leur ordre d'entrée. Ce système nous offre une gestion très simple des retours multiples des fonctions. Chaque variable à retourner est ajoutée dans la file outParams dans le corps de la fonction appelée.

Schéma d'exécution	
WHILE	C#

`(func A B C)`	`inParams.Enqueue(A)`
	`.Enqueue(B)`
	`.Enqueue(C);`
	`func(inParams, outParams);`
	
	`func (Queue<BinTree> inputs, Queue<BinTree> outputs){`
	`[…]`
	`outputs.Enqueue(ret);`
	`}`

b. Affectations & Expressions

Lors d'une affectation (potentiellement multiple), il faut avant tout évaluer les expressions de droite avant de les affecter aux variables de gauche. On utilise alors des variables temporaires pour évaluer la valeur de ces expressions étape par étape, chacune correspondant à une variable nommée de la forme Y0.

Schéma d'exécution	
WHILE	C#

`A,B:=(func S T), (hd X)`	`inParams.Enqueue(S)`
	`.Enqueue(T);`
	`func(inParams, outParams);`
	`Y0 = outParams.Dequeue();`
	`inParams.enqueue(X);`
	`BinTree.head(inParams, outParams);`
	`Y1 = outParams.Dequeue();`
	`A = Y0;`
	`B = Y1;`

c. Expressions conditionnelles

Que ça soit dans une boucle ou un branchement conditionnel, les expressions servant de condition sont évaluées de la même manière. On récupère l'arbre binaire résultant de l'expression et on vérifie s'il est égal à "`nil`" ou non grâce aux méthodes de la librairie BinTree détaillée au point II.2.b.

Schéma d'exécution	
WHILE	C#

[…]
while ((tl A) or (hd B)) do
  […]
od
[…]
	[…]
	Y0 = BinTree.tail(A);
	Y1 = BinTree.head(B);
	Y2 = BinTree.evaluate("OR", Y0, Y1);
	while(Y2.isTrue()){
	  […]
	  Y0 = BinTree.tail(A);
	  Y1 = BinTree.head(B);
	  Y2 = BinTree.evaluate("OR", Y0, Y1);
	}
	[…]

d. Le prélude

Afin de générer un programme C# compilable, il faut bien évidemment y ajouter des prérequis. Ce sont les imports et les identificateurs de classe. Ce prélude est constant, peu importe le programme while traduit.

Schéma d'exécution	
WHILE	C#

[…]	using System;
	using System.Collections.Generic;
	
	Namespace BinTreeProject{
	  class Program {
	  […] //Fonctions et postlude
	  }
	}

e. Le lanceur (Postlude)

En While, la dernière fonction du fichier constitue le point d'entrée du programme. Lors de la traduction, nous générons un lanceur qui exécute cette fonction particulière en premier, avec éventuellement des paramètres. Ce lanceur est le `main()` C# du programme généré. Contrairement au prélude, il est généré dynamiquement selon le programme While à traduire. En effet, pour des soucis de compréhension, nous conservons les noms des fonctions lors de leur traduction. Donc pour pouvoir exécuter la fonction principale, il faut mettre à jour le lanceur avec le nom du point d'entrée et le nombre de paramètres passé lors de l'exécution du programme généré.

Schéma d'exécution

WHILE	C#

[…]
 	
function run:
read X
%
  […]
%
write X
	public void run(Queue<BinTree> input, Queue<BinTree> output){
	  […]
	}
	
	public void main(argc, args[]){
	  //Conversion des arguments en BinTree
	  X0 = convertStrToBinTree(args[0]);
	  //Même action pour les éventuels autres paramètres
	  run(inParams,outParams);
	  outParams.dequeue();
	  //Récupération du résultat de la fonction
	}

2. Validation du projet

Afin de valider les différentes fonctionnalités composant notre projet, nous avons mis en place de batteries de tests unitaires. Chacun de ces ensembles de tests correspond à une partie spécifique du développement de notre projet (Pretty printer, code 3 adresses, table des symboles, ..).

Elles nous ont permis de vérifier, dans un premier temps, le bon fonctionnement de nos développements, en validant le résultat produit. Puis dans un deuxième temps, de vérifier la robustesse et la conformité de notre code lors de modifications.

a. Validation du Pretty Printer

Dans un premier temps, et ce en parallèle de la réalisation du Pretty Printer, nous avons mis en place une quarantaine de tests unitaires permettant de vérifier la validité du traitement réalisé.

Pour ce faire, nous avons établi deux types de tests. Les premiers consistent à Pretty Printer un fichier d'entrée (appelé original), dont le contenu concerne un fragment spécifique de la grammaire et dont la mise en page est chaotique. Le résultat produit (appelé résultat) est alors validé en comparant son contenu avec un oracle (un fichier correspondant au résultat attendu).

Les deuxièmes consistent à vérifier, via la récupération d'une exception, qu'un fichier original ne correspondant pas à la grammaire WHILE lève une exception.

b. Validation du code 3 adresses

Dans un second temps, nous avons mis en place une batterie de 31 tests unitaires concernant le code 3 adresses. Ceux-ci permettent de vérifier le bon fonctionnement et la bonne traduction d'un fichier original en WHILE en un ensemble d'étiquettes comportant du code 3 adresses. Ce code 3 adresses sert d'intermédiaire à la traduction finale en C#.

Pour valider cet aspect, nous avons réalisé deux types de tests.

Les premiers effectuent une traduction de code spécifique WHILE (nop, affectation simple ou multiple, cons, list, hd, tl , structure de contrôle, …) vers du code 3 adresses. Puis ils vérifient, via une méthode que nous avons créée, que le nombre d'étiquette produit correspond à ce qui est attendu.

Les seconds tests étaient plus difficiles à mettre en place. Ceux-ci évoluaient au fil de l'évolution de nos choix de traductions et des difficultés rencontrées. Ils consistaient à traduire du code spécifique WHILE en code 3 adresses et à vérifier étiquettes par étiquette, code 3 adresses par code 3 adresse que le résultat obtenu était celui voulu.

c. Validation de la table des symboles

En parallèle de la mise en place de tests unitaires concernant le code 3 adresses, nous avons réalisé 18 tests unitaires permettant de valider le code lié à la mise en place de la table des symboles.

Chacun de ces tests permettent de vérifier que la table des symboles contient bien tous les éléments attendus et qu'ils sont corrects. Entre autre, ils testent que la table des symboles produits contient toutes les variables, couplées à leur nombre d'occurrence, peu importe leur emplacement dans le programme, et que le nombre d'entrée et de sortie de chaque fonction

sont correctement renseignés. Chaque test prend en considération un programme While différent, chacun implémentant une commande ou une expression spécifique.

Afin de vérifier le comportement global de la table des symboles, et non sur des fonctionnalités précises, nous avons préparé des tests sur des programmes While plus complexes. Pour cela, nous écrivons la table de symboles générée dans un fichier, au format xml. Puis nous comparons l'xml créé pour chaque test au fichier attendu, à la façon d'un Oracle.

d. Tests de charges

Afin de tester la robustesse de notre compilateur, nous avons mis en place des tests de charge permettant de constater le comportement de ce dernier dans des cas d'utilisation plus poussées. Chaque test est lancé dix fois afin d'en obtenir une moyenne, où nous avons au préalable retiré les extremums.

Dans un premier temps, nous avons mis en place un test en largeur : nous avons réalisé un programme qui crée un nombre donné de fonctions basiques à la suite dans un même fichier. Puis nous calculons le temps de d'exécution de notre compilateur lorsqu'il compile ce fichier. Nous obtenons ainsi l'évolution du temps d'exécution de notre compilateur en fonction de la taille du fichier à compiler.

Dans un second temps, nous avons effectué un test en profondeur : pour cela, nous avons implémenté un programme qui crée un nombre donné de boucles « `while` » imbriquées. De la même manière, nous calculons le temps moyen d'exécution du compilateur afin d'observer l'évolution de celui-ci.

D'après nos résultats[1], la taille du fichier à compiler ne pose que peu de problème au compilateur. Mais lorsqu'il s'agit d'un programme imbriquant plus d'une centaine de commandes du langage, nous obtenons une erreur résultant du remplissage de la pile d'Xtext, lors du parsage du fichier.

[1] Voir les résultats complets à cette adresse : Tests de charge – Projet C#

II. Architecture logicielle

1. Les fichiers utiles au projet

Les fichiers indispensables et fixes :

- **whc.exe** : Script de lancement du compilateur avec différentes options
- **whpp.exe** : script de lancement du pretty printer avec différentes options
- whc.jar : jar exécutable lançant le compilateur
- whpp.jar : jar exécutable lançant le pretty printer
- whc_lib : répertoire contenant les librairies (.jar) utilisées par whc.jar
- whpp_lib : répertoire contenant les librairies (.jar) utilisées par whpp.jar

Tous ces fichiers doivent être placés dans le même répertoire sous-peine de ne plus fonctionner (ou impliquant une modification des .jar et des sources des scripts.)

- BinTree.cs : Source C#. Librairie libwh décrite au point b.

Ce fichier peut être déplacé mais demande une légère modification du source du script whc.cpp afin de recompiler l'exécutable final.

Certains fichiers sont auto-générés lors de l'exécution du compilateur :

- Le fichier de sortie .cs (source C#) indiqué en option au compilateur (out.cs par défaut)
- Le fichier exécutable .exe indiqué en option au compilateur (out.exe par défaut)
- Le fichier source While indiqué en option au pretty printer (sth.whpp par défaut)

2. Organisation des packages

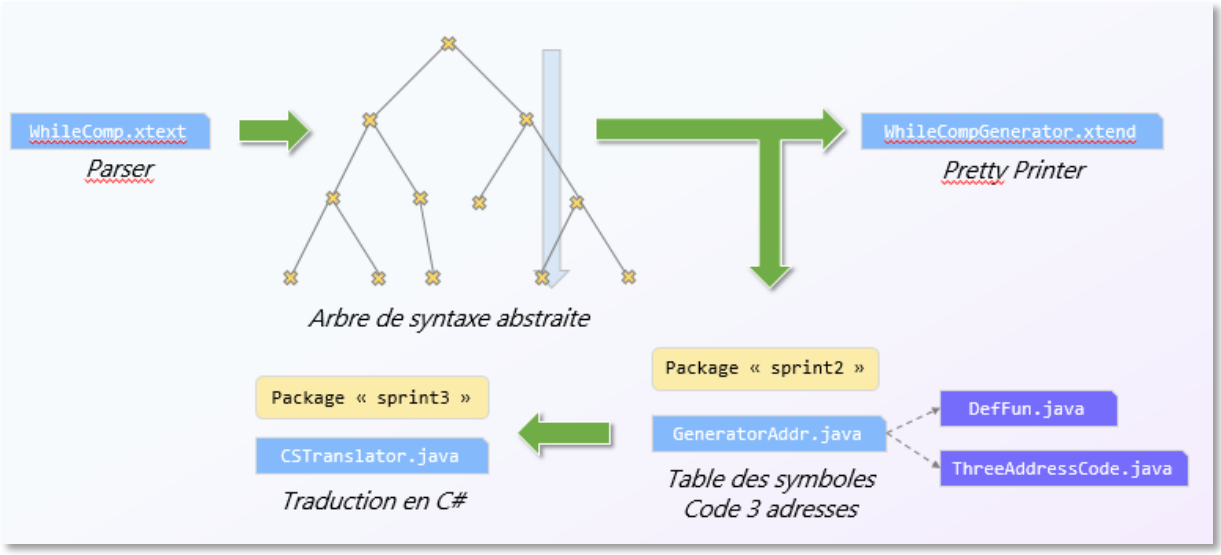


Figure 1 - Schéma de l'organisation générale des fichiers et classes importants du projet.

a. La grammaire Xtext

La première chose que l'on a dû créer pour ce projet est la grammaire du langage While (Fichier `WhileComp.xtext`). Le langage en lui-même n'est pas compliqué mais on a dû s'y reprendre à plusieurs reprises pour obtenir le résultat voulu. Cette grammaire nous permet de récupérer les différentes parties du programme (parser) et de le décomposer comme nous le souhaitons pour que l'on puisse générer le code 3 adresses correspondant à chaque morceau du programme.

b. Le générateur de code intermédiaire

La classe `GeneratorAddr` lance le parcours en profondeur de l'arbre de syntaxe abstraite généré par le parser Xtext. A chaque parcours d'un nœud, nous mettons à jour la table des symboles et créons, si besoin, un quadruplet de code 3 adresses en utilisant respectivement les classes `DefFun` et `ThreeAddressCode`. Ces classes permettent aisément la navigation des données générées lors du parcours de l'arbre. Il suffit de disposer du nom d'une fonction pour y récupérer toutes les informations la concernant dans la table des symboles et le code 3 adresses.

c. Tests

Pour les tests, nous avons utilisé JUnit. JUnit est un framework de test unitaire compatible avec Java. Celui-ci nous a permis d'écrire des scripts de test propre à chaque composante de notre système. Ainsi, vous pouvez trouver dans le répertoire "`testUnitaire`" du projet "`esir.compilation`", les packages "`prettyPrintTest`" et "`traductionTest`" correspondant respectivement à la partie "PrettyPrint" du projet et à la partie "traduction".

Dans le package "`prettyPrintTest`", on trouve le script de test correspondant nommé "`PrettyPrintTest.java`" ainsi que 3 dossiers, un premier contenant les fichiers à traiter (pretty-printé), un deuxième contant les oracles (résultats attendus) et un troisième contenant les fichiers après leur traitement.

Le package "`traductionTest`" est similaire dans sa structure. Nous avons des scripts de tests pour le code 3 adresses "`Code3AdressesTests.java`", la table des symboles "`TableDesSymbolesTest.java`", la traduction en C# "`SchemaDeTraductionTest.java`" et l'exécution "`ExecutionTest.java`". Chacun dispose d'un dossier contenant les fichiers sources en WHILE qui seront utilisés en entrés des tests dédiés.

3. La « Libwh » avec BinTree

Afin de permettre la traduction d'un programme While en C#, nous avons dû créer une librairie `BinTree.cs`. Cette librairie contient le type de variable qui sera utilisé dans le programme C# final. Elle contient toutes les méthodes utiles à l'utilisation de ces variables telles que `cons`, `list`, `head` ou `tail`. Nous pouvons aussi évaluer ces BinTree via la fonction `evaluate()` avec trois paramètres différents : EQ (=?), AND et OR. Nous nous basons sur cette librairie pour la traduction des instructions While, elle est donc indispensable au bon fonctionnement du code généré en C#.

4. Scripts de lancement

Nous avons créé plusieurs scripts de lancement pour nos différents exécutable que ce soit pour le Pretty Printer ou pour le compilateur. Ces scripts ont été fait en C++ afin de pouvoir les compiler pour les différents OS. Notre compilateur étant fait en Java, il est donc multi-plateforme donc il y avait un fort intérêt à faire un script de lancement lui-même multi-plateforme.

Les scripts de lancement permettent d'exécuter les différents programmes avec des options personnalisant l'exécution. Chaque script de lancement dispose d'un manuel d'utilisation, une présentation des options et la présentation des auteurs.

Le script `whpp.exe` se contente de lancer le .jar associé au Pretty Printer avec le fichier à formater en paramètre et le chemin vers le fichier de sortie.

Le script `whc.exe` permet de lancer le compilateur via le .jar associé, créant ainsi un fichier de sortie C# (.cs) puis, via l'option –e de créer l'exécutable du programme While traduit en C#. Nous compilons avec ce programme .cs la librairie BinTree décrite ci-dessus à chaque traduction, afin qu'elle soit disponible dans le .exe final.

5. Gestion des code 3@

La production du code intermédiaire est technique et assez complexe à mettre en place. La partie la plus difficile a été la gestion des expressions. Nous avons décidé d'utiliser des arbres pour gérer l'ensemble des expressions. Voir annexe 1 pour une explication schématique du fonctionnement interne de l'arbre des expressions en montrant sur un exemple.

Notre arbre empile les différents symboles et variable, puis une fois l'expression terminée, simplifie l'arbre en sous arbre et génère du code 3 adresses. Pour les fonctions acceptant des listes de fonctions comme "cons" nous avons décidé de faire des "push" afin de recréer cette liste avec seulement un code 3 adresses. Le système inverse avec des pop en plus pour les symboles "fonctions" définis dans le programme While à compiler.

Les quadruplets générés sont mis dans une liste d'étiquette, elle-même gérée avec une pile dynamique pour gérer les imbrications.

Toutes ces informations sont accessibles depuis une Map qui associe les étiquettes générées avec les quadruplets qu'elles contiennent. Lors de la génération du code, nous parcourons simplement la liste de ces codes 3 adresses et les traduisons selon leur nature.

III. Bilan de gestion de projet

1. Étape du développement

a. Nos outils

Nous nous sommes organisés de la manière suivante :

- Utilisation de Git via le logiciel Source Tree. Il était indispensable pour nous d'utiliser Git. En effet, cet outil nous permet de pouvoir travailler à plusieurs sur le projet sans risquer quoi que ce soit (perte de code). Nous pouvons facilement savoir qui a fait quoi récemment ou bien décidé d'effectuer une branche à part pour un travail spécifique, etc.
- Trello est un très bon outil de gestion de projet. Il permet de savoir où nous sommes rendus dans le projet, ce qui est à faire, ce qui est en train d'être fait et par qui ainsi que ce qui a été fait. Nous pouvons avoir facilement un aperçu de l'avancé de notre travail sur le projet.

Ces outils nous ont donc permis de pouvoir gérer notre projet plus facilement.

b. Nos forces et nos faiblesses

Même si nous avons eu quelques difficultés quant à l'utilisation de SourceTree, il nous a quand même permis (ainsi que Trello) de bien organiser notre projet et cela a plutôt bien marché. D'un autre côté, nous n'avons pas pris assez de temps pour faire des réunions de mise à niveau du travail, pour que chacun ait connaissance des avancés des autres. Cela nous aurait permis de plus facilement comprendre les parties du projet où certains d'entre nous ont moins travaillé.

2. Rapport d'activité individuels

a. Rapport d'activité de Florent Catiau-Tristant : chef de projet

En tant que chef de projet, mon premier rôle fut d'organiser l'équipe. Ainsi, dès le démarrage du projet, je donnai des directives à l'ensemble des membres de l'équipe pour préparer l'environnement de travail (installation des IDE, plugins, etc.). De plus, j'ai opté pour l'utilisation des outils de gestion tels que Trello et Git (via le logiciel SourceTree, après une recommandation de Dylan) lors du projet, une décision rapidement approuvée par tous.

Lors de la première phase du projet, je participai à l'élaboration de la grammaire While en Xtext. Bien que celle-ci fût en partie construite, il nous fallait une version commune pour tout le groupe. J'aidai ensuite mes camarades à la conception du Pretty Printer, en Xtend. Afin de maintenir toute l'équipe dans la même direction, je travaillais sur toutes les parties à la fois (documentation, développement et tests).

A la suite du développement du Pretty Printer, je séparai l'équipe en binômes afin d'être le plus agile possible sur les différentes parties du projet. Dylan et Quentin s'occupaient des tests, Alexis et Mehidine du code 3 adresses, Yoann du schéma de traduction et moi-même pour la table des symboles. J'ai donc pu concevoir l'entièreté de la table des symboles, en ajoutant le parcours de l'arbre de syntaxe abstraite, les structures de données formant la table des symboles, et des différentes méthodes de vérification et de tests (format xml, vérification des usages des symboles,…). Je gardais toujours un œil sur l'avancement général du projet, aidant par moment Dylan à la conception des tests et à l'intégration de la table des symboles au code 3 adresses avec Alexis et Mehidine par exemple.

Durant la dernière partie du projet, je mis à jour les points importants du projet jusqu'ici mis à l'écart : les scripts de lancement du projet, les tests dits négatifs, préparation de la démonstration finale et conformité avec les spécifications clients. Par ailleurs, j'ai beaucoup pris part à la correction des bugs présents dans le programme, notamment en remontant la localisation des erreurs aux personnes ayant développées les parties concernées par les bugs.

Pendant toute la durée du projet, je m'occupais de préparer nos supports de présentation pour nos séances « d'avancement projet ». Je prenais des notes sur les remarques énoncées lors de ces séances et, en général, je me chargeais de faire les corrections. J'organisais aussi les réunions « critique » lors des phases de débuggage et de conception importantes.

b. Rapport d'activité de Dylan Béhêtre : responsable des tests

Lors du développement du projet, mon principal rôle a été de réaliser les tests unitaires.

J'ai ainsi défini l'architecture du test : la technologie, comme la construction de ceux-ci.

J'ai ensuite réalisé les tests unitaires du Pretty Printer. Pour ce faire, j'ai reçu l'aide de Quentin pour la réalisation des oracles et l'aide de Quentin_et Florent_pour les tests négatifs (les tests vérifiant que ce qui ne doit pas passer, ne passe pas).

Une fois passé sur la deuxième partie du projet, j'ai réalisé les tests dédiés au code 3 adresses dans leur ensemble ainsi que les tests dédiés à la table des symboles. Concernant les tests de la table des symboles, Florent a renforcé mes tests en établissant une vérification de la table via une comparaison à une structure en XML.

J'ai également participé aux réflexions en amont sur le schéma de traduction et le code 3 adresses. J'ai suivis les avancés des membres de mon groupe afin d'en comprendre les enjeux et les objectifs. Cela m'a également permis de réaliser les tests appropriés.

De plus, j'ai participé à la rédaction du rapport de fin de projet, notamment la mise en forme et le cadrage. J'ai participé à la réalisation de l'introduction produit avec l'aide de Quentin la partie validation du projet, et parlé de la partie test au sein de la partie architecture logicielle.

J'ai également réalisé avec l'aide de Quentin et Florent la mise en place de l'option « - test » qui permet pour la partie Pretty Printer ou la partie traduction d'effectuer les tests correspondants via une ligne de commande.

J'ai finalement mis en place un système permettant de réguler l'affichage en console ainsi que les chemins d'accès aux différents fichiers selon que le code soit exécuté via le JAR ou dans Eclipse.

c. Rapport d'activité de Yoann Boyere

Mon activité a commencé lors du sprint1 en écrivant la grammaire while dans le projet xText. J'ai ensuite aidé le reste du groupe pour effectuer le pretty printer.

Ensuite, au début du sprint2, je me suis attelé à rédiger avec Quentin le schéma de traduction. Celui-ci a continué à être modifié lors de ce sprint ainsi que le 3ème. J'ai de mon côté aidé Mehidine à continuer la partie "pretty printer".

Assez vite, me rendant compte que les deux autres importantes parties (table des symboles et code 3 adresses) étaient bien avancées et maîtrisées par Florent et Alexis, je suis très vite passé sur le dernier sprint en commençant la librairie en c# : BinTree.cs. Cette classe est obligatoire afin de pouvoir créer les variables (des arbres) ainsi que les évaluer (cons, list, and, or, etc).

Une autre partie importante du sprint était la traduction du code 3 adresses en C#. Alexis m'a donné les bases de cette traduction afin de mieux appréhender son travail sur le code 3 adresse. A partir de ça, j'ai donc travaillé sur la traduction vers le code C# (création des fonctions, utilisation des reads et writes, le main, passages des arguments,etc). Mehidine et Alexis ont eux aussi contribué à cette traduction.

Enfin, j'ai aidé à la résolution des bugs concernant la génération du code final, faire le script de compilation du fichier final c# et à la rédaction de ce document.

d. Rapport d'activité de Alexis Brault

Ma contribution dans le projet a commencé par la production de l'exécutable whpp avec ses différents paramètres.

Je me suis ensuite greffer au code de Florent pour générer le code 3 adresses. Toute la première partie, je l'ai réalisé seul puis Méhidine est venu m'aider pour compléter les parties du code 3 adresses pour lequel je n'avais pas eu le temps d'avancer.

D'autre part, j'ai conçu le code permettant de générer le code C# à partir du code 3 adresses généré auparavant. Puis j'ai donné les clés de ce nouveau code à Yoann pour qu'il puisse mieux l'appréhender le compléter avec son avancement sur la libwh.

De plus, j'ai essayé d'aider au mieux que je pouvais Dylan pour qu'il puisse réaliser ses tests même si cet échange n'a pas été des plus simple, difficulté de ma part à expliquer ce que j'ai fait et surtout le vulgariser à la juste valeur pour qu'il puisse faire les tests nécessaires.

Et enfin, j'ai participé à la rédaction de ce livrable, à la communication dans le groupe ainsi que la mise en place des outils ainsi que la mise à jour de leur contenu.

e. Rapport d'activité de Mehidine Chupeau

J'ai commencé à prendre part au projet dès le sprint 1 en programmant le Pretty Printer avec l'aide de Yoann qui a écrit la grammaire du langage While. J'ai par la suite dû améliorer la grammaire et le Pretty Printer lors de ce sprint 1 car il y avait des problèmes que l'on n'a pas su voir mais que les tests de Dylan ont révélés.

A la fin du Pretty Printer, Florent, Alexy et Yoann avaient déjà commencé le sprint 2, je me suis donc tourné vers Alexis qui n'avait pas encore fini le code 3 adresses pour l'aider à le terminer. J'ai donc pu programmer le code 3 adresses des expressions booléennes ainsi que le code 3 adresses des boucles for, foreach, while et if. Comme nous travaillons en méthode agile, à la fin de chaque fonction que je codais dans le code 3 adresses, je codais en même temps sa traduction en C# dans le fichier CS_translator.java.

Durant toute la durée du projet, je suis resté en contact avec les tous les membres du groupe pour que je puisse leurs apporter de l'aide dans les différents bugs rencontrés mais plus particulièrement avec Dylan qui était le responsable des tests pour qu'il puisse me dire s'il existe des problèmes dans le code que j'ai écrit.

Vers la fin du sprint 3, j'ai passé du temps avec Alexis, Florent et Yoann pour que l'on puisse régler des petits détails dans le code 3 adresses et la génération de code et notamment la classe BinTree en C# pour que le compilateur soit fonctionnel le jour de la présentation.

f. Rapport d'activité de Quentin Olivier

Dans un premier temps, j'ai participé avec l'aide de Dylan à la réalisation des tests unitaires visant à contrôler le bon fonctionnement du livrable du sprint 1, c'est à dire le Pretty Printer. Alors que Dylan mettait en place le système de tests, je me suis occupé de la rédaction des différents fichiers destinés aux oracles.

Ensuite, au lancement du sprint 2, j'ai participé avec le reste de l'équipe à l'élaboration du schéma de traduction, qui a été mis à jour tout au long de ce sprint afin d'être raccord avec l'implémentation du code 3@ et de la table des symboles.

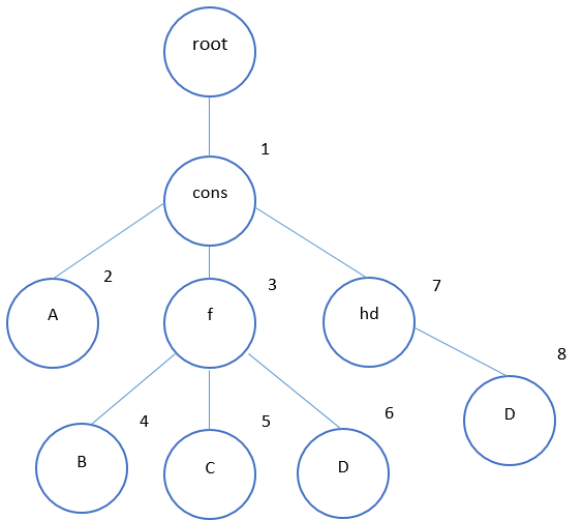
Pour le sprint 3, j'ai repris le fichier source qu'Alexis avait réalisé pour créer l'exécutable du Pretty Printer afin de réaliser l'exécutable du Compiler. Florent l'a ensuite repris pour le compléter avec les fonctionnalités manquantes et ainsi finaliser l'exécutable.

Enfin, j'ai aidé Dylan à l'implémentation de l'option de lancement des tests dans les deux exécutables : cela consiste brièvement à créer un moyen de lancer les tests JUnit via une option -test sur les exécutables.

3. Annexes

a. Annexe 1 : schéma de l'arbre des expressions

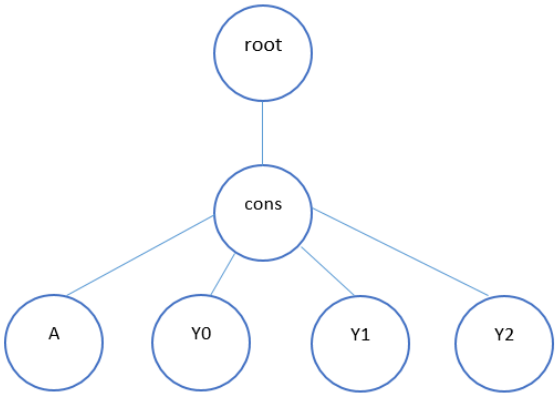
Soit l'expression : `A := (cons A (f B C D) (hd D))`
, f étant une fonction a 3 entrées et 2 sorties. La numérotation correspond à l'ordre d'entrée dans l'arbre.



Voici les étapes de simplification :

`(f B C D)` est devenu `Y0` et `Y1`; `(hd D)` est devenu `Y2`; soit `A := (cons A Y0 Y1 Y2)`

ExprTree	Code 3 adresses généré :

	`<PUSH, _, B, _ >`
	`<PUSH, _, C, _ >`
	`<PUSH, _, D, _ >`
	`<CALL f, _, _, _ >`
	`<POP, Y0, _, _ >`
	`<POP, Y1, _, _ >`
	`<HD, Y2, D, _ >`

	`<PUSH, _, A, _ >`
	`<PUSH, _, Y0, _ >`
	`<PUSH, _, Y1, _ >`
	`<PUSH, _, Y2, _ >`
	`<CONS, Y3, _, _ >`