[COMP] Rapport projet **WHILE** vers **CPP**

Sébastien Blin, Pierre-Henri Collin, Kevin Ledy, Alexandre Quettier, Benjamin Vion

Description générale

Ce rapport décrit comment nous avons choisi d'implémenter la traduction du langage **WHILE** vers **CPP**.

Schéma de traduction

Dans notre projet, nous visions donc la conversion du langage **While** vers le langage **C++**. Pour parvenir à cette conversion, nous avons du établir très tôt une liste des correspondances entre les structures **While** et les les composants que nous offre le **C++**. Toutefois, avant cette transcription, il a été nécessaire de passer du code **While** vers du code 3 adresses afin de simplifier la tâche.

Correspondance While → C++

Structure de données

Comme nous le savons, en **While**, il n'existe qu'un seul type de structure de données. Cette structure est un arbre binaire. Il a donc été décidé dès le départ que nous représenterions cette structure de données par une classe **C++** *BinTree*. Cette classe va donc regrouper toutes les méthodes nécessaire à l'exécution d'un code **While**. On trouve ainsi des méthodes permettant de tester si le *BinTree* est à *nil*, si c'est un symbole ou encore une variable. On trouve également des méthodes d'accès sur les attributs de la classe et qui permettent donc d'accéder aux attributs *head* et *tail*. En revanche, on trouve dans cette classe *BinTree* une méthode qui

correspond à une commande de **While**, la méthode *cons*. Cette méthode va donc prendre 2 *BinTree* pour n'en former qu'un (les deux *BinTree* en entrée devenant les *head* et *tail* du nouvel arbre). Cette méthode ainsi que les précédentes sont déclarés en *static* en **C++**, pour que ces méthodes soient utilisables sans instancier un seul objet puisqu'elles caractérisent un *BinTree* en général et non un objet.

Structures de contrôle

Pour ces traductions, nous reprenons simplement les structures existantes en **C++**. Un *if* **While** est donc traduit vers un *if* **C++**. En utilisant les méthodes permettant de tester des conditions (*isNil* notamment), on exécute la boucle le nombre de fois nécessaires.

Code 3A

Avant de procéder à la génération du **C++**, nous avons réalisé une transcription du **While** en code 3 adresses. Nous présentons dans la suite de ce document le code 3 adresses que nous avons choisi pour nos structures.

Tableau de correspondances

While	Code 3 adresses
X = nil	< nil, X, _, _ >
nop	< nop, _, _, _ >
X = (cons A B)	< cons, v_1, A, B > < :=, X,v_1, _ >

While	Code 3 adresses
X = (cons A B C)	< cons, v_1, B, C > < cons, v_6, A, v_1 > < :=, X,v_6, _ >
X = (hd Y)	< hd, X, Y, _ >
X = (tl Y)	< tl, X, Y,_ >
X = Y =? Z	< =?, X, Y, Z >
X := Y	< :=, X, Y, _ >
X := (foo Y)	< call, X, foo, Y >
if cond then codeThen else codeElse	< IF l_0 l_1, _, cond, _ >
while cond then code od	< WHILE l_0, _, cond, _ >
for cond then code od	< WHILE l_0, _, cond, _ >
foreach elem in ensemb do cmds od	< FOREACH l_0, _, elem, ensemb >

Justifications des codes 3 adresses

Le code 3 adresses que nous avons choisi est un code que nous avons voulu simple et efficace. Pour les instructions simples comme les affectations, le *nop*, les tests d'égalités, nous avons repris le code 3 adresses que M. Ridoux nous avaient montrés en cours de compilation.

Justification des codes 3 adresses :

- **< CONS, v_1, B, C > < CONS, v_6, A, v_1 > < :=, X, v_6, _ >** : Cette représentation découle des choix précédents pour notre pretty printer ou nous avions limiter un cons à seulement 2 opérandes. Il fallait donc automatiquement remplacer `cons A B C` par `cons A (cons B C)`. De la même façon, ici nous isolons des paires de variables pour faciliter la génération de code.
- **< CALL, X, foo, Y >** : Dans cette représentation, nous isolons en premier, les variables où l'on va stocker le résultat de la méthode (ici X) et en second les variables qui sont passées en paramètres (ici Y).
- **< IF l_0 l_1, _, cond, _ >** : Les labels *l_0* et *l_1* contiendront respectivement le code du *Then* et le code du *Else*. Ils seront exécutés suivant l'évaluation de la condition *cond*
- **< WHILE l_0, _, cond, _ >** : Exactement le même principe que pour le *if*, l'évaluation de la condition conditionne l'exécution du code situé au label *l_0*. Le code 3 adresses du *while* est également utilisé pour représenter un *for* par commodité.
- **< FOREACH l_0, _, elem, ensemb >** : Ici encore, la même utilisation des labels est faite tant qu'on garantie l'évaluation de la condition.

Architecture logicielle

Ce projet est découpé en 2 parties distinctes. La première, la plus grosse est composé de tout le code servant à traduire le **WHILE** vers du **CPP** Ces parties sont présentes dans les dossiers *whileComp**. Celle-ci est divisée en plusieurs sous parties :

- *whileComp/src/* contient les interfaces d'entrées, c'est-à-dire *whpp.java* pour le pretty printer et *whc.java* pour le compilateur
- *whileComp/src/org/xtext/example* contient le fichier de définition de la grammaire

- *whileComp/src/org/xtext/generator* contient les générateurs comme le PrettyPrinter, le générateur de code 3 adresses (*ThreeAddGenerator.xtend*) et le générateur de code C++ (*CppGenerator.xtend*) ainsi que le fichier appellé lors de la sauvegarde d'un fichier sur une instance Eclipse (*WhileCppGenerator.xtend*).
- *whileComp/src/SymbolTable* contient nos classes utiles pour la génération et les tests.
- *whileComp.tests/src/* contient nos tests unitaires du PrettyPrinter, de la table des symboles et du ThreeAddGenerator (Nos tests seront développés plus loin).

La seconde partie contient le code servant lors de l'éxécution d'un programme **WHILE**, c'est-à-dire la **libWh**. Cette bibliothèque est présente dans le dossier *CPP* et contient le fichier *BinTree.h* et *BinTree.cpp*. Cette bibliothèque peut-être liée de 2 façons à notre programme **WHILE**. Soit en la donnant comme bibliothèque à g++, soit comme fichier du projet. Nous avons opté pour la seconde méthode. Ainsi nous avons juste à appeller `g++ -o test BinTree.* FICHIERWHILETRADUIT.cpp -std=c++11` notre fichier traduit incluant cette bibliothèque (`#include "BinTree.h"`)

Pour récapituler :

1. Nous appellons notre programme via `java -jar whc.jar FICHIER.wh -o FICHIER.cpp`
2. La classe *Whc* récupère le contenu du fichier d'entrée et le donne à traiter au *ThreeAddGenerator*
3. Ce générateur va alors vérifier que le fichier est correct et générer le code 3 adresses du code **WHILE**. Il redonne alors à la classe *Whc* une liste de Fonctions (un objet fonction contenant la table des symboles, et une liste de quadruplet), une map entre les noms de fonctions du code **WHILE** et leurs équivalents **Cpp**, la liste des labels créés (contenant des quadruplets) et les erreurs trouvées dans le fichier **WHILE**.
4. La classe *Whc* passe alors ces résultats à la classe *CppGenerator*. Ce générateur va alors traduire les quadruplets générés en code **Cpp**, inclure les

bonnes bibliothèques, générer la fonction main et retourner le code généré
5. Enfin la classe *Whc* écrit le code généré dans le fichier de sortie et compile avec g++

Les outils de productivité

Git

Pour gérer les différentes versions du Compilateur, nous avons utilisé l'outil **Git**. Ainsi nous avions accès à la fois à la documentation et aux sources du projet. Mais ce qui nous intéressait particulièrement était les *Issues* qui permettent de créer des posts où l'on peut débattre sur un sujet, parler de bugs, etc.

Nous nous en sommes donc servis en faisant différents types d'issues :

- ToDoList : pour énumérer les tâches à faire et qui prenait quoi
- Comptes-Rendus-Réunions : prises de notes ou commentaires émisdurant les réunions d'avancement du projet, qui nous permettaient après coup de revenir dessus
- Des exemples de traduction de WHILE vers C++ pour toujours savoir vers où nousallions. Nous avons essayé de couvrir le maximum de cas
- Autres : ces issues étaient plutôt temporaires car le débat portait soit sur la grammaire au départ du projet, soit d'un problème technique, d'une manière de concevoir, etc

D'ailleurs **Git** nous a aussi aidé dans le versionnage car plusieurs bugs ont pu être débusqués et réparés en regardant les différences entre deux versions. Par conséquent, **Git** a été un outil très important afin de mener à bien ce projet, tant sur l'aspect programmation que conception ou débats et questions.

Par contre, il y a de nombreuses fonctions utiles de **GIt** que nous n'avons pas utilisé

comme donner des noms de versions, ou le développement sur plusieurs branches (ce qui aurait été plus logique). Enfin on aurait modifié la gestion des TODO List en les écrivant plus précisément depuis le début, ça nous aurait évité d'oublier des trucs commes `=?` ou not.

Validation du projet

Couverture des tests

Notre stratégie de tests s'articule autour de 4 grandes parties qui suivent l'avancée du projet : tout d'abord nous avons un fichier de test (chemin : *whileComp.tests/src/whileComp/tests*) qui a pour but de tester le bon fonctionnement du PrettyPrinter (*PrettyPrinterTest.xtend*). Ce fichier contient plusieurs types de test :

- Vérifier que le PrettyPrinter affiche correctement le code While conformément aux exigences de Mr. Ridoux. Nous avons testé l'affichage pour la structure **while**, les **cons**, les **list**.
- Vérifier que si on applique le PrettyPrinter sur lui-même, on obtient le même résultat
- Vérifier `whpp-1(f) = whpp-1(whpp(f))`
- Vérifier `whpp(f) = whpp(whpp-1(f))`
- Réaliser des stress tests (en largeur, en longueur et en profondeur)

Le second fichier de test a pour mission de vérifier le bon fonctionnement de la table de symbole.

Le troisième fichier de test a pour but de tester le passage du code while au code 3 adresses :

Pour ce qui est du passage du code 3 adresses au code **C++**, nous n'avons pas

réalisé de fichier de test. Nous avons testé *à la main* que la traduction se réalise correctement. Pour cela, nous avons créé un répertoire (demo) qui contient un ensemble de fichiers **While** avec des fonctions simples et d'autres plus compliqués, puis après avoir lancé le générateur de code **C++**, on compare le résultat avec la spécification que nous avons défini dans le schéma de traduction.