

Le 2 mai, 2020

**TP5 - Langage machine**

LOG3210

Éléments de langages et compilateur

Remis à

Mme Doriane Olewicki

Philippe Maisonneuve – 1959052

Nicole Joyal – 1794431

Table des matières

[Introduction 1](#_Toc38976258)

[Implémentation 1](#_Toc38976259)

[Diagramme UML 1](#_Toc38976260)

[Tests 1](#_Toc38976261)

[Conclusion 2](#_Toc38976262)

# Introduction

Dans le cadre du ce travail pratique, nous avons été convoqués à rédiger un code machine qui et en tenant compte des contraintes de mémoire. Premièrement, sans contraintes de mémoire nous avons rédigé le code machine associé au code intermédiaire. Avec le code machine, nous avons implémenté un graphe d’interférence et la coloration de celui-ci. En nous servant du graphe colorié, nous avons géré l’assignation de registres avec l’aide des *spill* quand nécessaire. Le tout a été utilisé lors de la simulation du code Fibonacci. Nous allons expliquer ci-dessous notre stratégie d’implémentation, avec un diagramme UML comme support et terminer avec un résumé de nos tests et la justification de ceux-ci.

# Implémentation

Afin d’implémenter la génération du code machine, nous avons commencé par l’ajout d’un nouvel objet *MachLine* dans l’attribut *CODE* à chaque fonction *visit* pertinente. Nous nous sommes servis des listes *LOADED* et *MODIFIED* pour savoir quand ajouter un *load* du registre au code machine. Une fois que le code machine sans contraintes de mémoire à été rédigé dans la liste *CODE*, nous avons procédé à la génération et à la coloration du graphe d’interférence.

Pour générer le graphique, nous avons ajouté un attribut de type *HashMap<String, ArrayList<String>>*, appelé grapheInterference. Cet attribut représente chaque nœud (registre) et ses voisins dans une structure de type *Map*. L’avantage de l’utilisation du *Map* est l’unicité des clés; qui représentent dans notre cas les nœuds. Aussi, la facilité d’accès à un nœud particulier est avantageuse durant le *spill* afin de rapidement mettre à jour le graphe (gestion du *stack* dans l’algorithme de coloration). L’attribut grapheInterference à été utile surtout lors de la coloration de celui-ci. Cependant, nous nous sommes servis d’un autre *HashMap<String, String>* pour gérer la coloration. Cela permettait, comme pour le *Map* du graphe d’interférence, de maintenir l’unicité des clés et rapidement accéder aux nœuds et leur couleur respectif.

# Diagramme UML

# Tests

# Conclusion