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

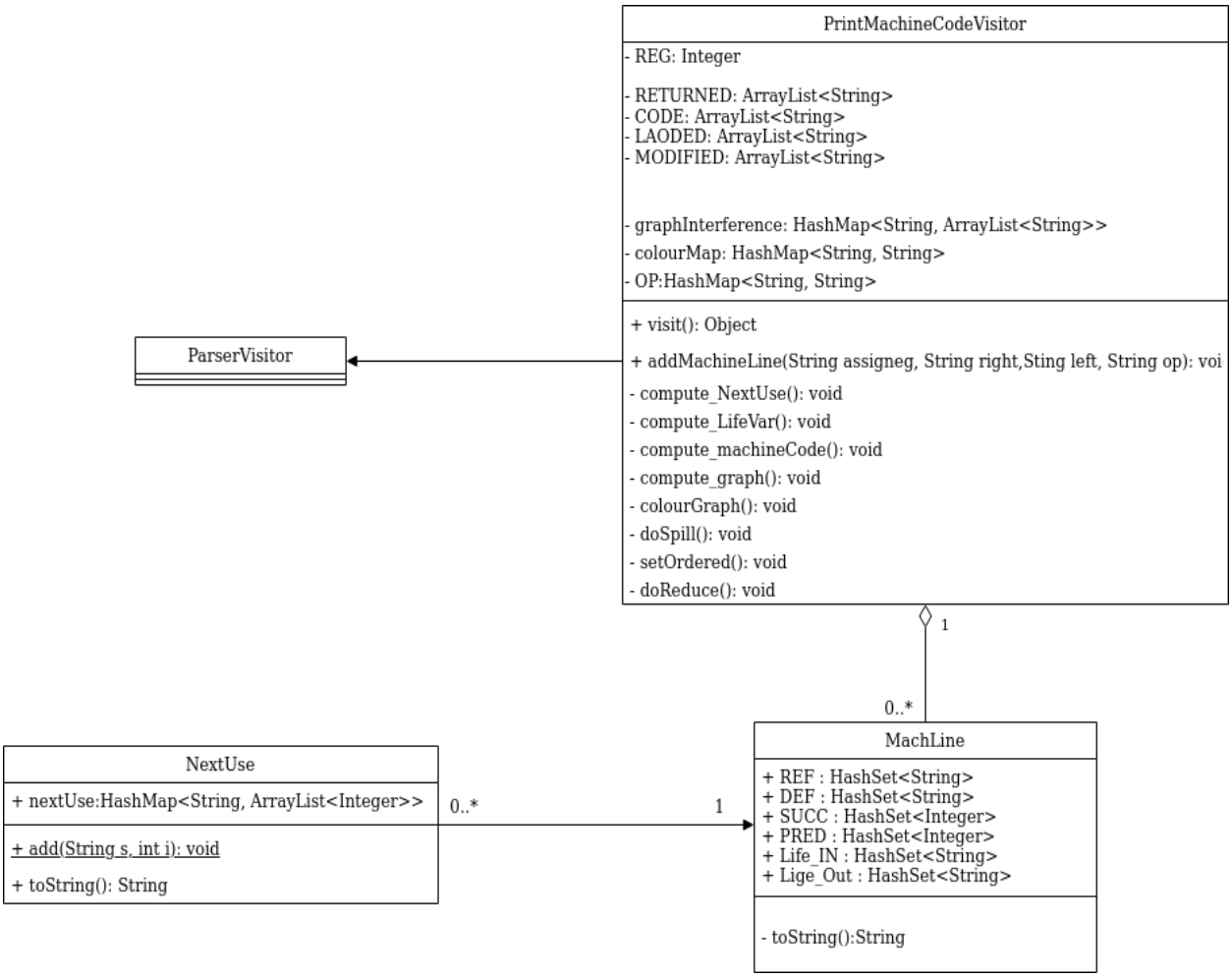
Dans le cadre de ce travail pratique, nous avons été convoqués à rédiger un code machine en tenant compte des contraintes de mémoire. Premièrement, sans contraintes de mémoire nous avons implémenté le code machine associé au code intermédiaire. Avec le code machine, nous avons implémenté un graphe d'interférence 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 servi 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 respective.

Diagramme UML



Tests

Nous avons réalisé 4 tests différents afin de nous assurer du bon fonctionnement du code.

Le premier test servait à vérifier si notre fonction permettant de simplifier le code en supprimant les opérations inutiles fonctionnait bien. Pour ce faire, nous nous sommes contentés de coder tous les cas que nous prenions en compte dans notre fonction de réduction. C'est-à-dire les cas où nous ajoutions 0 à un nombre et les cas où nous multiplions par 1.

Le second test servait à vérifier si notre compilateur allait utiliser des registres non nécessaires dans certains cas. Plus spécifiquement dans le cas où deux variables ne sont pas utilisées dans la même opération et ne devraient donc pas être voisines l'une de l'autre dans le graphe, mais son rendu égale l'une à l'autre. Comme les deux variables étaient utilisées sur la même ligne, il aurait été facile de croire que les deux variables devaient être stockées dans deux registres différents, mais ce n'est pas le cas, car le même registre peut être utilisé sur la même ligne en entrée et en sortie. Nous voulions être certains que notre compilateur en tiendrait compte.

Le troisième test est assez simple, il sert uniquement à nous permettre de voir le fonctionnement du système vérifiant la durée du vide des variables. Il a été conçu pour donner une sortie un résultat très facile à lire afin que nous puissions facilement suivre ce qui se passe.

Le dernier test servait quant à lui à vérifier le bon fonctionnement du do_spill(). On a donc volontairement testé un cas où le nombre de registres était trop petit pour pouvoir colorer le graphe sans appeler cette fonction. Pour le code en tant que tel, nous nous sommes contentés de réutiliser celui du test 3.

Conclusion

Bref, l'objectif de ce laboratoire était de traduire du code source en code machine avec la considération des contraintes de registres. Afin d'accomplir cela, nous nous sommes servis surtout des structures *Map* et *List* pour la gestion de l'état. Nous avons implémenté plusieurs tests afin de vérifier les contraintes de registres. La plus grande difficulté pour nous a été de gérer les *spill*. Si nous avions à refaire ce TP, nous aurions consacré plus de temps à la gestion des *spill*.