LOG3210 - Éléments de langages et compilateurs

Hiver 2020

Polytechnique Montréal

TP #5

Langage Machine

Soumis à **Doriane Olewicki** le 23 avril 2020

Par :

Sami Tamer Arar, 1828776

Jaafar Kaoussarani, 1932805

Table des matières

Description détaillée de l’implémentation 2
Génération du code intermédiaire 2
Coloration de graphe 2
Description des tests 3
Diagramme UML de notre implémentation 4

Description détaillée de l'implémentation

Génération du code intermédiaire

Tout d'abord, nous analysons les lignes qui utilisent l'opérateur « = », soit l'assignation directe, les opérations unaires (négation) et les assignations avec opération (addition, soustraction, multiplication, division). Pour le premier, nous générons un « *ADD* » avec 0 comme élément de gauche (équivalent à *x = 0 + y*). Pour le deuxième, nous générons un «*MIN*» avec 0 comme opérande de gauche (équivalent à *x = 0 - y*). Pour le troisième, nous vérifiant le type de l'opération et ajoutant la ligne de code correspondante. Nous pouvons utiliser « *ADD* » pour les additions, « *MIN* » pour les soustractions, « *MUL* » pour les multiplications et « *DIV* » pour les divisions (Exemple: *a = b + c* devient *ADD @a, @b, @c*). Dans tous les cas, si une variable à droite du « = » n'a pas encore été chargée, alors on génère une ligne « *LD* » et on ajoute cette variable à l'ensemble « *LOADED* ». De plus, si une variable est modifiée, on l'ajoute à l'ensemble « *MODIFIED* ». Finalement, pour la partie « *return* », si une variable a été modifiée, on génère une ligne « *ST* ». Les ensembles « *life* » et « *next use* » sont calculés exactement comme décrits dans l'énoncé.

Coloration de graphe

Après avoir calculé le « *next use* » de chaque ligne, nous construisons le graphe. Pour ce faire, nous itérons sur l'ensemble « *Next_OUT* » de chaque ligne et ajoutons tous les pointeurs à la liste de noeuds de la classe *Graph*. Ensuite, nous réitérons encore une fois sur l'ensemble « *Next_OUT* ». Pour chaque ligne, si deux noeuds différents se trouvent dans cet ensemble, alors nous ajoutons une arête entre ces noeuds. Afin d'éviter des vérifications booléennes complexes, nous utilisons une « *HashSet* » afin d'éliminer les doublons, d'où le *«override»* des méthodes *«equals»* et *«hashcode»* dans la classe *Edge*.

Pour la coloration du graphe, nous avons utilisé la méthode expliquée dans l'énoncé pour nous guider. Premièrement, nous clonons l'ensemble des noeuds et arêtes dans des structures temporaires afin de pouvoir simuler le vidage et le repeuplement du graphe. Pour le vidage, nous cherchons deux ensembles de noeuds, soit les noeuds ayant le plus de voisins, mais inférieur à *k* (soit le nombre de registres) et les noeuds ayant le plus de voisins, mais supérieur à *k*. S'il y a au moins un noeud dans l'ensemble plus petit que *k*, on retire ce noeud du graphe et le mettons dans le *stack* avant de recommencer le tout jusqu'à ce que le graphe soit vide. Cependant, s'il n'y a aucun noeud inférieur à *k*, alors on fait une tentative de *spill* telle que décrite dans l'énoncé. S'il est possible de *spill*, alors on régénère le graph et recommençons le coloriage. Sinon, nous continuons l'algorithme de coloriage comme si de rien ne s'était passé. Finalement, après avoir vidé le graphe, nous appelons *pop* sur le *stack* et régénérons le graphe noeud par noeud. À chaque boucle, nous assignons une couleur au noeud tel que ses voisins ont une couleur différente. Ces couleurs correspondent aux registres qui remplaceront les pointeurs dans le code machine.

Description des tests

- *test_all_ops*: Ce test consiste en l'utilisation des quatre opérations, soit la multiplication, la division, l'addition et la soustraction, dans le même bloc de code. Cela permet de démontrer que le code est capable de bien distinguer entre ces opérations. De plus, puisque le cas "full" utilise 5 registres (R0-R4), nous testons aussi une version à 3 registres de ce test afin de démontrer encore plus la justesse du coloriage.

- *test_big_return* (Robustesse): Dans ce test, nous utilisons un gros *return*. Cela nous permet de tester la robustesse et aussi de démontrer que le code est capable d'assigner un grand nombre de registres. Ainsi, afin de tester encore plus la robustesse, nous créons une version de test qui n'utilise que 3 registres. Cela démontre aussi la flexibilité et la justesse du coloriage (Nous voyons en effet que le code est capable de le faire avec un seul registre).

- *test_long_var_name* (Robustesse): Dans ce test, nous utilisons des noms de variables excessivement longs. Cela permet de tout simplement démontrer que le code n'est pas confus par des noms de variables longs.

- *test_many_ref*: Ce test consiste en l'utilisation d'une variable donnée à divers endroits dans le code (par exemple, la variable «a» est utilisée aux lignes 3, 4, 6,et 11, la variable «b» aux lignes 3, 4, 7, 11 et 14, etc.). Cela demande l'utilisation de plusieurs registres, d'où le fait que ça nous permet de démontrer la flexibilité de la coloration ainsi que sa capacité à considérer les utilisations à long terme des variables. De plus, puisque le cas *full* utilise 8 registres (R0-R7), nous créons une version à 5 registres de ce test afin de démontrer encore plus la justesse du coloriage.

- *test_normal_case*: Ce test consiste en la création d'un cas d'utilisation «typique», soit en utilisation des noms de variables et des opérations raisonnables (notez cependant que le nom des variables et les opérations sont complètement arbitraires). Ainsi, nous démontrons tout simplement que le code fonctionne bien dans des conditions «normales».

- *test_numbers_only* (Robustesse): Ici, nous n'utilisons que des nombres lors de l'assignation des variables initiales (soit a, b, c, d). Cela permet de tout simplement démontrer que le code gère bien les nombres de grosse et de petite taille.

- *test_unary_full*: Puisque les tests fournis n'utilisent pas les opérations unaires (dans notre cas, juste la négation), nous créons un test afin de vérifier que cette partie du code est fonctionnelle et que le coloriage se fait correctement.

Notez que nous avons aussi tester le code fournis avec l'énoncé avec le script Fibonacci pour les blocs à 3 registres, 5 registres et sans limite. Nous avons essayer plusieurs valeurs entre 0 et 50 et avons toujours obtenue les bons résultats, d'où notre confiance en la justesse du code.

Diagramme UML de notre implémentation