LOG3210 – Éléments de langages et compilateurs

TP Final

Département de génie informatique et de génie logiciel

École Polytechnique de Montréal



Soumis par

Roman Zhornytskiy (1899786)

Jeannorasinh Tong (1745133)

Groupe : 01

Soumis à Doriane Olewicki

Hiver 2020

**1. Description de l’implémentation**

1.1 Description de la génération du graphe d’interférence

Notre graphe d’interférence est un HashMap <String, ArrayList<String>> où le String représente le nœud et le ArrayList<String> représente les nœuds qui sont connecté à ce nœud (ce sont les arrêts du noeud).

On construit ce graphe en parcourant, chaque ligne du CODE et chaque nœud du Next\_OUT. Si notre graphe d’interférence ne contient pas le nœud du Next\_OUT, alors on l’ajoute à notre graphe. Ensuite, on parcourt une autre fois chaque nœud du Next\_OUT pour ajouter tous les arrêts du nœud.

En même temps qu’on construit le graphe, on construit notre List<String> nodes qui contient tous les nœuds en ordre alphabétique que nous allons utiliser plus tard.

1.2 Description de la coloration du graphe d’interférence

1. On crée une copie identique de notre graphe d’interférence qui sera utilisé plus tard (nommé : grapheInterferance2) et on crée le Stack pour empiler les nœuds.

**Géneration de la Stack**

1. Tant que notre graphe d’interférence n’est pas vide :
   1. On sélectionne le nœud ayant le nombre de voisin le plus proche et inférieur à k en parcourant les nœuds en ordre alphabétique. Si rien n’a été trouvé, alors c’est le premier nœud de la List « nodes » qui va être sélectionné.
   2. On enlève le nœud sélectionné (« nodeToStack ») du graphe et on parcourt tous les nœuds du graphe pour enlever les instances de ce nœud dans la liste des autres nœuds (on enlève ses arrêts). On push le nœud sur la Stack.
2. On crée le colorMap<String, Integer> (le string représente le nœud et Integer représente le numéro ou la couleur du nœud)
3. Tant que la Stack n’est pas vide :
   1. color = 0. On pop un nœud et on ajoute ce nœud dans notre graphe d’interférence.
   2. On parcourt tous les nœuds du graphe et on ajoute ses arrêts avec l’aide de la copie identique du graphe.
   3. **Coloration se fait en 2 parties :**
      1. On crée une liste <Integer> « voisins » ordonnée en ordre croissant. Cette liste contient les valeurs de couleurs de ses voisins.
      2. On parcourt en ordre croissant cette liste et si la valeur du voisin est égale à color, alors on fait color++.
   4. On ajoute le nœud qu’on a pop et sa valeur de color dans colorMap.

**Changement du CODE**

1. On parcourt chaque ligne du CODE
   1. Pour chaque ligne on remplace les pointeurs qui commencent par « @ », par « R + color » dans colorMap.

1.2 Description de la génération du code intermédiaire

**2. Description des tests**

En parcourant les tests fournis dans les sources du tp nous avons constaté que pas toutes les opérations n’ont été testé. Nous observons que les deux seules opérations utilisées sont l’addition, la multiplication et la division. Et c’est à partir de cette observation que nous avons décidé de rajouter un test afin de compléter le jeu de tests fourni. Le test que nous avons rajouté traduit un bloc de code avec une opération de soustraction (MIN).

Les blocs fournis traduisent les codes suivants :

* Block1\_full :

t = d\*(b + a) + c\*b;

a = d\*b + c\*a;

b = t;

* Block2\_full :

t = d\*(2\*c + d);

c = c\*c + d\*d;

d = t;

i = i / 2;

Le bloc que nous avons rajouté traduit le code suivant :

* Block3\_full :

t = d\*(b - a) + c\*b;

a = d\*b + c\*a;

b = t;

tester réduction  
a = a

a = a \* 1

a = a / 1

**3. Diagramme UML**

A screen shot of a computer

Description automatically generated